AD-A052 731 CHARLES STARK DRAPER LAB INC CAMBRIDGE MA F/6 9/2
JOVIAL STRUCTURED DESIGN DIAGRAMMER (JSDD). VOLUME III. PROGRAM--ETC(U)
FEB 78 G GODDARD, M WHITWORTH, E STROVINK F30602-76-C-0408
UNCLASSIFIED R-1120-VOL-3-PT-1 RADC-TR-78-9-VOL-3-PT-1 NL

1 OF 2
AD
A052 731

2

RADC-TR-78-9, Vol III, Part 1 (of four)
Final Technical Report
February 1978

JOVIAL STRUCTURED DESIGN DIAGRAMMER (JSDD) Volume III
Program Description. Part 1

G. Goddard
M. Whitworth
E. Strovink

The Charles Stark Draper Laboratory, Inc.

Approved for public release; distribution unlimited.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

Because of the size of this volume, it has been divided into four parts. Part 1 contains pages 1/2 - 123, 649 - 657, Part 2 contains pages 124 - 344, Part 3 contains pages 345 - 592, Part 4 contains pages 593 - 648.

RADC-TR-78-9, Vol III, Part 1 has been reviewed and is approved for publication.

APPROVED: Donald Van Alstine
DONALD VANALSTINE
Project Engineer

APPROVED: Wendall C Bauman
WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss
JOHN P. HUSS
Acting Chief, Plans Office

18 RADC   19 TR-78-9-VOL-3-PT-1

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-78-9, Vol III, Part 1 (of four)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER 9
~~4. TITLE (and Subtitle)~~ 6 JOVIAL STRUCTURED DESIGN DIAGRAMMER (JSDD). ~~Program Direction~~ Volume III. Program Description. Part 1.		~~5. TYPE OF REPORT & PERIOD COVERED~~ Final Technical Rep~~ort~~. Sep~~tember~~ 76 — Oct~~ober~~ 77
		6. PERFORMING ORG. REPORT NUMBER 14 R-1120 - VOL-3-PT-1
7. AUTHOR(s) 10 G. Goddard, M. Whitworth E. Strovink		~~8. CONTRACT OR GRANT NUMBER(s)~~ 15 F30602-76-C-0408
9. PERFORMING ORGANIZATION NAME AND ADDRESS The Charles Stark Draper Laboratory, Inc. 555 Technology Square Cambridge MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS P.E. 62702C J.O. 55811412
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIM) Griffiss AFB NY 13441		12. REPORT DATE 11 Feb~~ruary~~ 1978
		13. NUMBER OF PAGES 132 12 138p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same 16 5581 17 14		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DDC

APR 17 1978

F

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald VanAlstine (ISIM)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Structured programming	Preprocessor
Structured design diagram	Flowcharter
Structured extension	JOVIAL J3
Parse	Invocation diagram
Parser generator	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The report presents a detailed description of the JOVIAL Structured Design Diagrammer program implementation for purposes of maintaining or modifying the system.

DD FORM 1 JAN 73 1473   EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

408 386   JOB

Acknowledgement

This report was prepared by The Charles Stark Draper Laboratory, Inc., under Contract F30602-76-C-0408 with the Rome Air Development Center at Griffis Air Force Base.

Especial credit is due Margaret Hamilton, who pioneered principles of Structured Programming at Draper Laboratory. Saydean Zeldin originally suggested the symbology implemented in the output of the JOVIAL Structured Design Diagrammer. Thanks should go also to William Daly, who created the Structured Design Diagrammer for the HAL language (currently being used on the NASA Space Shuttle project). The authors are indebted to Victor Voydock for his invaluable assistance in implementing a complete MULTICS user interface which was used successfully for the duration of the JSDD implementation. The authors are also grateful to J. Barton DeWolf whose many suggestions were of great assistance throughout this effort.

PROGRAM DESCRIPTION

This document was produced to satisfy the requirements of contract number F30602-76-C-0408 with the Rome Air Development Center. It is one of four companion volumes:

* JOVIAL Structured Design Diagrammer (JSDD) Report Summary

  This document is a summary of the contents of the JSDD Final Report.

* JOVIAL Structured Design Diagrammer (JSDD) Final Report

  This volume presents the design techniques for implementing the JSDD and describes the use of Structured Design Diagrams.

* JOVIAL Structured Design Diagrammer (JSDD) Program Description

  This volume presents a detailed description of the program implementation for purposes of maintaining and/or modifying the JSDD.

* JOVIAL Structured Design Diagrammer (JSDD) User's Manual

  This volume presents the user's view of the JSDD along with user options and other information about running the program.

TABLE OF CONTENTS

1. Introduction

In recent years, the digital computer software industry has directed considerable effort toward the development of design and implementation methodologies to ensure the sufficiency, reliability, and maintainability of software systems. The most widely known product of this effort is the loosely defined set of design and programming practices called "Structured Programming" (see references 1, 2, and 3).

Structured Programming does not constitute a complete software development methodology. Rather, it is a collection of general guidelines for use by software designers and implementors. As such, it provides no uniform approach to system design and offers no method of evaluating system sufficiency with respect to requirements or design. Despite these shortcomings, adherence to the Structured Programming principles can be of great assistance in producing software systems which are reliable and intellectually manageable.

The techniques of Structured Programming are sufficiently general to allow system developers a tremendous amount of stylistic freedom. However, the generality of the techniques has made the development of a standard approach to software analysis extremely difficult. The prototype JOVIAL Structured Design Diagrammer (JSDD) is the first component of an integrated software analysis and documentation system which will address itself to this task.

The JSDD is an automated analysis and documentation system which produces two types of diagrams: Structured Design Diagrams (SDDs) and Invocation Diagrams. SDDs provide a graphic display of program control logic. Invocation Diagrams are a display of a software system's functional (calling) structure.

The JSDD processes digital computer programs written in either JOVIAL J3 or Extended JOVIAL J3. Extended JOVIAL J3 is standard JOVIAL J3 as specified in reference 4 plus the structured extensions to JOVIAL J3 (see JSDD Final Report) which are based upon Reference 5.

This document describes the internal structure of the JSDD programs, with the objective of providing enough detailed implementation information to allow their easy modification.

An effort has been made to provide a top-down approach to this detailed structural information; thus, it is hoped that a high-level understanding of the JSDD can be acquired by an untrained reader without any necessity on his/her part to read through needless technical detail. By the same token, the detailed information is still available (at lower section levels) for systems engineering personnel.

2. Computer Definition

The JOVIAL Structured Design Diagrammer is designed to run on all Honeywell Information Systems Inc. Series 6000 computers providing at least one disk drive, one line printer, and 96K of user memory in addition to the memory or separate input/output devices required by the GCOS operating system identified in Section 3.

3. System Description

The JOVIAL Structured Design Diagrammer is designed to run under the Honeywell Information Systems Inc. Series 60 Level 66 and Series 6000 General Comprehensive Operating Supervisor (GCOS) Version I/G.

4. Program Description

The JSDD consists of three programs, organized as two conceptual "passes." The first pass is referred to (interchangeably) as the "Design Diagram Database Generator" (DDDG) or "Phase 1". The second pass consists of two programs, one which draws Structured Design Diagrams (SDDs) and another which draws Invocation Diagrams. The first program is referred to (interchangeably) as the "Design Diagram Generator" (DDG) or "Phase 2". The second program is always called the "Invocation Diagrammer." (see Figure 4-1)

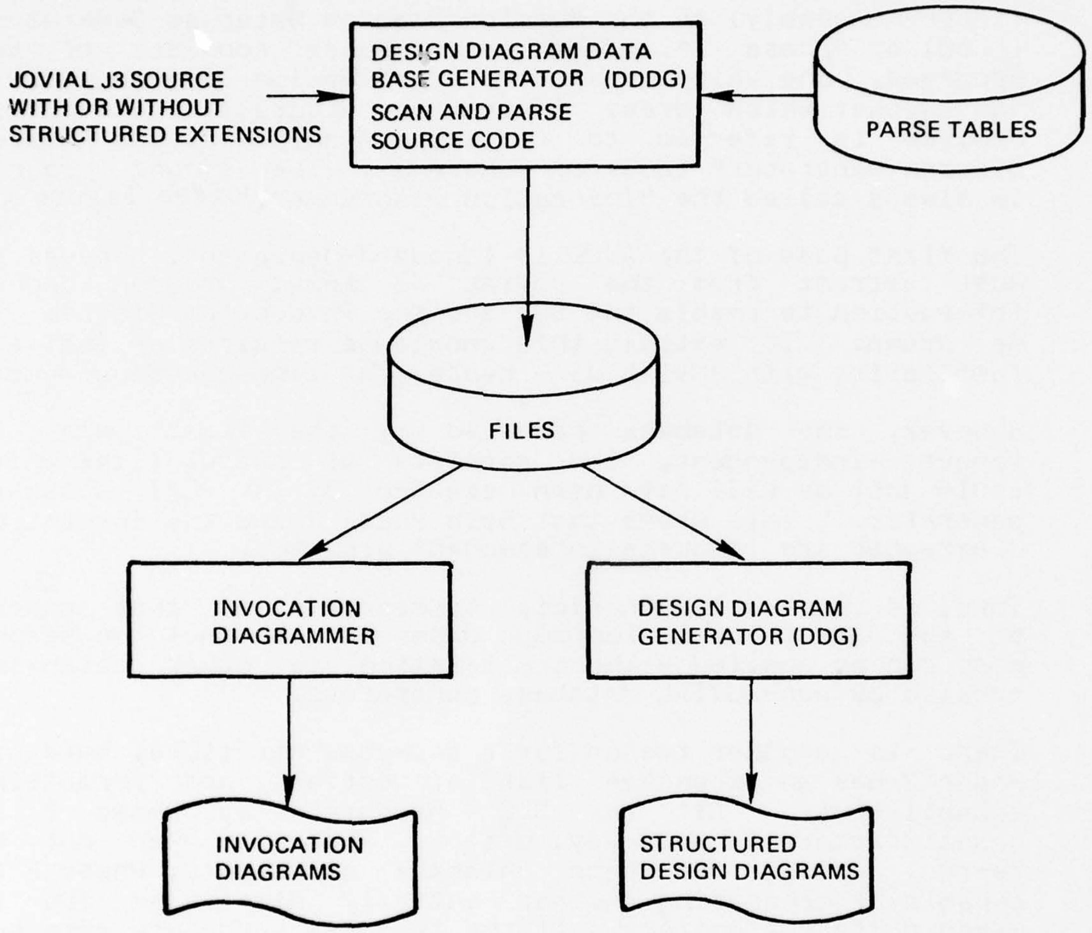
The first pass of the JSDD is language-dependent, because it must extract from the JOVIAL J3 input program enough information to enable the SDD and the Invocation Diagram to be drawn. To extract this knowledge requires an intimate familiarity with JOVIAL J3 - hence, the language dependence.

However, the database produced by the first pass is language-independent, and consists of output files which could just as well have been created by a PL/I database generator. This means that both Phase 2 and the Invocation Diagrammer are language-independent programs.

Thus, it is clearly beneficial to separate out that aspect of the JSDD which is language dependent, so that the second pass can be applied without alteration to other databases created by non-JOVIAL database generators.

There is another reason for a two-pass structure, however: Phase 2 has an extensive list of options and formatting capabilities. If an SDD produced by Phase 2 is unsatisfactory in some way, oftimes Phase 1 need not be re-run. Using the same database as before, Phase 2 is capable of generating another entirely dissimilar SDD if given different options. If the two-pass structure were not present, Phase 1 would have to be re-run as well, wasting computation resources.

Section 4 is organized as follows: Section 4.1 discusses the extended string package developed for use in the JSDD; Sections 4.2-4.3 describe Phase 1; Section 4.4 discusses the Phase 1 output data files; Sections 4.5-4.6 cover the structure and operation of Phase 2; Section 4.7 describes the Invocation Diagrammer; and Section 4.8 contains instructions for compiling the JSDD programs.



JOVIAL J3 SOURCE WITH OR WITHOUT STRUCTURED EXTENSIONS
DESIGN DIAGRAM DATA BASE GENERATOR (DDDG) SCAN AND PARSE SOURCE CODE
PARSE TABLES
FILES
INVOCATION DIAGRAMMER
DESIGN DIAGRAM GENERATOR (DDG)
INVOCATION DIAGRAMS
STRUCTURED DESIGN DIAGRAMS

Figure 4-1. Two-Pass Construction of the JSDD

4.1. The String Package

The need for string manipulation subroutines is acute in compiler-like programs, whose only purpose is the transformation of one series of strings into another. The string capability provided with JOVIAL J3 is insufficient for such an application. Therefore, it was necessary to create separate string-handling routines to provide an elementary string capability.

Section 4.1.1 discusses the inadequacy of the JOVIAL string capability, Section 4.1.2 discusses the format of "extended" strings, Section 4.1.3 introduces the string routines, Section 4.1.4 contains routine descriptions, and Section 4.1.5 lists compool and internal variables.

4.1.1 JOVIAL String Manipulation

JOVIAL J3 offers character string variables of static maximum length which can be compared and assigned. Character string literal constants can also be used. To provide a slightly more advanced capability, the built-in function BYTE is capable of extracting a sub-string of a character variable, given a zero-based index into the string and a length argument.

The fundamental problem with this capability is that the <u>current</u> *length of a given character string is not available.* For that matter, neither is the (declared) maximum length, except during compilation. Additionally the character strings are <u>right</u>-justified. This forces tedious subtraction calculations to get at the "real" text (using BYTE).

A typical problem encountered with JOVIAL strings is the inability to assign spaces to a character string. The statements:

```
        ITEM AA H 150$
        AA = 7H(       )$
```

are meaningless - it is impossible to determine whether AA contains 7 blanks, 10 blanks, 1 blank, or 150 blanks (leading blanks in text strings are also lost in the same way).

In an environment where complex substrings and concatenations occur as often as arithmetic expressions, this awkwardness is unacceptable.

4.1.2 Extended String Format

The solution to the string-handling problem is to encode the length of the character string inside the string variable. Then user-defined functions can perform various advanced string operations such as SUBSTR and CONCAT.

It is necessary first to define a standard maximum length for all character strings. This length is set to 144, based on standard line-printer page width and divisibility by 6 (word alignment) considerations. In order to uniquely identify them, the character "[" is placed in the first byte of the character strings. The next five bytes contain the length of the string. The next 144 bytes contain the string itself - thus, each string is declared with length 150. All text is left-adjusted in the 144 bytes of character string. For example:

```
JOVIAL:                "      ...                     HELLO"
                        0                                  149

EXTENDED STRING:       "[00005HELLO                ...      "
                        0                                  149
```

4.1.3 Routine Descriptions

The presence of descriptor-based character strings (strings with a built-in length attribute) immediately suggests a LENGTH function and a NIL character string (a string containing nothing). The format of the strings requires an OUT routine, which strips off the unwanted descriptors before outputting strings. SUBSTR (sub-string) and CAT (concatenation) routines are required by definition. A NULL function (to return 1 if a string is NIL) is provided, but is unnecessary because a character string can be initialized to NIL, and used for comparisons. A CNVERT (convert) routine is necessary to convert ordinary JOVIAL strings to the descriptor form. Although the extended strings are capable of representing varying-length blank strings, some thought will convince the reader that creating such strings is a non-trivial problem. Therefore, the SPACES function is provided, which creates descriptor-based varying-length blank strings.

The following code illustrates the use of some of the above functions:

(variables AA - DD below are declared as "H 150")

```
AA=7H(MARK 14)$
BB=1H($)$
CC=6H(......)$

DD=CAT(CAT(CAT(SUBSTR(AA,1,5),CC),CAT(SPACES(1),BB)),
   SUBSTR(AA,6,2))$

OUT(DD)$
```

The output is:

```
MARK ...... $14
```

4.1.4 String Routines Summary

The following sections describe the individual functions of the extended string package.

4.1.4.1 CAT

CAT's function is to return the string defined by the result of concatenating the second character string argument to the first.

CAT's operation is straightforward, utilizing the byte function to move characters from the second string to the first. Errors are detected by simple addition of the argument string descriptors (these strings are converted by a call to CNVERT if they were not already converted).

CAT errors cause a truncation of the result string (rightmost characters lost).

4.1.4.2 CNVERT

CNVERT's function is to return the converted version of its argument. If the argument string is too long (>MAXCOL characters), it is truncated (characters are lost starting at the rightmost). If the string is already converted, it is passed as the result.

CNVERT searches the passed string from left to right until it reaches a non-blank character. It then moves the text from this point to the end of the passed string into a temporary string. No more than MAXCOL characters are moved.

A descriptor is then placed in the leftmost six bytes of the result string, which have providentially been left blank during the move above.

4.1.4.3 LENGTH

LENGTH converts its argument string if necessary and then returns the value of the descriptor.

4.1.4.4 NULL

NULL simply converts its argument string (if necessary) and then returns 1 if the descriptor = 0.

4.1.4.5 OUT

OUT can write strings to JOVIAL file 12 or to the terminal. Terminal I/O is machine specific, as well as specific to the MULTICS GCOS environment, and will not be described. File 12 output is written in 80-byte records. The input string is converted if necessary, stripped of its descriptor, and written into two consecutive file 12 records if necessary. File 12 is the error file for both phases of the Structured Design Diagrammer.

4.1.4.6 SPACES

SPACES returns a converted string containing the number of blanks specified by its integer argument.

SPACES blanks its result string and inserts a descriptor equal to its integer argument. If the argument was less than 0, SPACES' result is undefined. If the argument was greater than MAXCOL, the result string is truncated.

4.1.4.7 SUBSTR

SUBSTR's function is to return the substring defined by a character string, a starting index, and a length. Characters are numbered from left to right, starting at 1. The length is the total length of the substring, including the beginning character pointed to by the starting index.

SUBSTR's task is straightforward, because the descriptor of the converted character string argument (if the argument is not converted, SUBSTR calls CNVERT to convert it) allows easy error checking. The BYTE function is used to handle the substring operation, and the passed length is incorporated into the result string descriptor.

SUBSTR errors cause a null string result.

4.1.5 String Package Compool and Internal Variables

```
common abc$
begin
 item const i 36 s p 10737418240$ ''= 6h([00000)''
 item const2 i 36 s p 21474836480$ ''twice const''
 item err h 150$ ''not used''
 item initst i 36 s p 1$ ''if set, out opens file 12''
 item ln1 i 36 s$ ''descriptor of overlay set 1''
 item ln2 i 36 s$ ''descriptor of overlay set 2''
 item ln3 i 36 s$ ''descriptor of overlay set 3''
 item ln4 i 36 s$ ''descriptor of overlay set 4''
 item ln6 i 36 s$ ''descriptor of overlay set 6''
 item maxcol i 36  s  p  132$  ''maximum  size  of  extended
strings''
 item rpterr i 36 s p 0$ ''flag directs error output''
 array sa1 25 h 6$ ''breaks overlay set 1 into words''
 array sa2 25 h 6$ ''breaks overlay set 2 into words''
 array sa3 25 h 6$ ''breaks overlay set 3 into words''
 item sf1 h 150$ ''parto of overlay set 1''
 item sf2 h 150$ ''part of overlay set 2''
 item sf3 h 150$ ''part of overlay set 3''
 item sf4 h 150$ ''part of overlay set 4''
 item sf6 h 150$ ''part of overlay set 6''
 item tc h 150$ ''temporary''
```

```
   item tcl h 150$ ''temporary''
   item tc6 h 6$ ''temporary''
   overlay sal=sfl=lnl$
   overlay sf4=ln4$
   overlay sa2=sf2=ln2$
   overlay sa3=sf3=ln3$
   overlay sf6=ln6$
   file zzzzzz h 12000 v 80 12$
     ''defines error output file attributes''
  end

  proc out(aa,cc)$
     item  aa h 150$ ''string whose contents is to be output''
     item  bb  h  150$  ''temporary  variable  which  holds
  intermediate text''
     item cc i 36 s$ ''if cc is 1, output to terminal;
                       otherwise to file 12''
     item dd h 80$ ''temporary variable - used to write
                       80-byte file12 records''


  proc substr(aa,first,num)$
     item aa h 150$ ''host string for substring operation''
     item  first  i  36  s$  ''index  of  first  character  of
  substring''
     item num i 36 s$ ''length of substring''
     item substr h 150$ ''result string''
```

```
proc cat(aa,bb)$

    item aa h 150$ ''leftmost string in concat''

    item bb h 150$ ''rightmost string in concat''

    item cat h 150$ ''result string''


proc cnvert(aa)$

    item aa h 150$ ''string to be converted''

    item cnvert h 150$ ''result string''

    item done i 36 s$ ''flag for internal while loop''

    item ii i 36 s$ ''temporary''

    item jj i 36 s$ ''temporary''


proc spaces(num)$

    item num i 36 s$ ''number of spaces in result string''

    item spaces h 150$ ''result string''


proc null(aa)$

    item aa h 150$ ''string to be tested for null  contents''

    item null b$ ''boolean result - =1 if string is null''


proc length(aa)$

    item aa h 150$ ''want to know length of this string''

    item length i 36 s$ ''length of aa is contained here''
```

4.2 Phase 1 Program Structure

4.2.1 Introduction

This section describes the structure and operation of the Design Diagram Database Generator (DDDG). Instructions on use of the Design Diagrammer are not appropriate here - they will be found in the CSDL JOVIAL Structured Design Diagrammer User's Manual.

Sections 4.2.1.1 gives a general overview of the DDDG. Section 4.2.2 contains a more specific description of DDDG structure and operation. Section 4.3.2 contains very detailed descriptions of specific DDDG functions and functional modules, while Section 4.3.1 gives the formats of all major internal DDDG databases.

This document is intended to provide a heirarchical description of the DDDG. For most purposes, Sections 4.2.1 and 4.2.2 should be sufficient for a high-level understanding of the program. Section 4.3 is available, however, for the systems programmer who needs to make additions or alterations to the DDDG.

This document should be read after the JOVIAL Structured Design Diagrammer User's Manual and in conjunction with the Structured Design Diagram of the DDDG (Section 7 of this volume).

4.2.1.1 Program Description

The Design Diagram Database Generator takes as input JOVIAL J3 source programs of the form specified in reference 4, and generates as output three output files, henceforth referred to as FILE 0, FILE 1, and FILE 2. FILE 0 contains a symbol table used by the Invocation Diagrammer. FILE 2 consists of formatted JOVIAL program text, separated by the DDDG into "statement units," a full list of which appears in Appendix B. FILE 1 contains pointers into the FILE 2 text and information about it. Files 1 and 2 are the primary flowcharter databases.

The DDDG is essentially a syntax-driven JOVIAL compiler with abbreviated semantic analysis and code-generation phases. Its symbol table mechanism is also primitive, in that it currently stores only the fact that certain procedures are called within the scope of other procedures. The code-generation and semantic analysis phases of the DDDG can be thought of as that code which produces the three output files.

Although the DDDG is an abbreviated compiler in some ways, it should not be inferred that a full-scale compiler could not be built easily using the DDDG as a base. In fact, a few code-generation-oriented changes to the JOVIAL J3 BNF description (with similar changes in the code-generation routines) could create the basic structure for a full JOCIT JOVIAL compiler.

4.2.2 Structure

4.2.2.1 Introduction

This section is intended to present the basic DDDG structure so that an understanding of DDDG operation can take place.

Section 4.2.2.2 discusses syntax-driven compiler structure, and Section 4.2.2.3 shows how the DDDG fits into this mold. Section 4.2.2.4 outlines the full DDDG architecture, as well as summarizing its modules. Section 4.2.2.5 gives an example-driven description of DDDG operation based on the data in Sections 4.2.2.2-4.2.2.4. This section relies heavily on abstract diagrams of high-level program structure, primarily because such diagrams illustrate most clearly interactions between procedures and data. In these diagrams, a double line represents a procedure call, a single line data flow, rectangular boxes procedures, and circles databases. This convention is followed throughout Section 4.2.2. In the text, database names occur in square brackets ([]) whenever their meanings are not obvious.

4.2.2.2 Syntax-Driven Structure

The DDDG is, as has been described in Section 4.2.1.1, built around a syntax-driven compiler structure. This type of structure is common in most modern compiler designs and is sufficiently flexible to accomodate all but the most arcane or needlessly complex languages.

The major components of a syntax-driven compiler are parsing tables, a parsing algorithm, and a synthesize routine. The tables are constructed, usually automatically, by means of an analysis of the language description, which is written in some form analogous to Backus-Naur form (BNF). These tables contain information about the legal sentences in the described language and give rules for associating certain types of strings with certain specific types of sentences. In this way, the tables act as a database for "understanding" sentences written in the language, since a correct parse will cause source strings to be "reduced" to more meaningful constructs.

Upon such a "reduction," the synthesize routine is called in to decide what action should be taken as a result of the new knowledge which has been gained. In most cases, synthesize will do very little; however, in many cases further semantic analysis or code generation must take place. The synthesis routine is generally implemented as a large "case" statement, with most of the cases either empty or containing small amounts of code.

There is of course one more function that is necessary - that is, breaking source input text into "tokens." A token is an irreduceable symbol in the chosen representation of the language. Typical tokens are reserved words in the language (if, proc, for) or user-supplied words (<identifier>, <number>). The task of analyzing the source input for such tokens is performed in most syntax-driven compilers by the scan routine, which itself contains enough wired-in information to pick out reserved words and user-defined variables, as well as to resolve any existing inadequacies in the language as implemented by the parsing routines. For instance, there are some language constructs not analyzable by existing automatic analysis techniques; but these can be dealt with by altering the description of the language to side-step the difficulty, and then massaging the tokens passed from the scanner to "fit" the new description. In this way, tokens that never actually exist in the language can be passed to the parsing routines as "dummy" tokens to avoid certain ambiguities.

The components described above are invoked or subsumed by a procedure which is central to the whole compilation process. This central procedure contains the parsing algorithm, and uses that algorithm to call on the other units of the syntax-driven compiler. A typical sequence of such calls might be READ (a parsing algorithm function), SCAN (get a new token), LOOKAHEAD (another parsing algorithm function), SCAN, APPLY (still another algorithm function), and SYNTHESIZE (to process the new knowledge from APPLY).

Figure 4-2 shows a basic syntax-driven compiler structure. COMPILATION'LOOP contains the parsing algorithm, and calls, when necessary, SCAN (to pick up new tokens) and SYNTHESIZE (to take action after a reduction). RECOVER is a special procedure used to recover from syntax errors. It adjusts the parse history (stored in [parse stack]) in such a manner as to allow compilation to continue in a reasonable fashion. RECOVER calls SCAN if it needs another token to make its parse history adjustment. [Tables] contains information used by COMPILATION'LOOP (and RECOVER) to make parsing decisions. [Parse stack], as mentioned above, is used by COMPILATION'LOOP to save the parse history, and [token data] contains information passed from SCAN to COMPILATION'LOOP.

It is not our purpose here to give a complete discussion of syntax-driven compilation; however, references 6 through 8 give a thorough coverage of the subject. Reference 6 deals with the theory of automatic grammar analysis, reference 8 describes the construction of a modern syntax-driven compiler and reference 7 gives the format of the parsing tables used by the DDDG, as well as the algorithms by which they were derived.

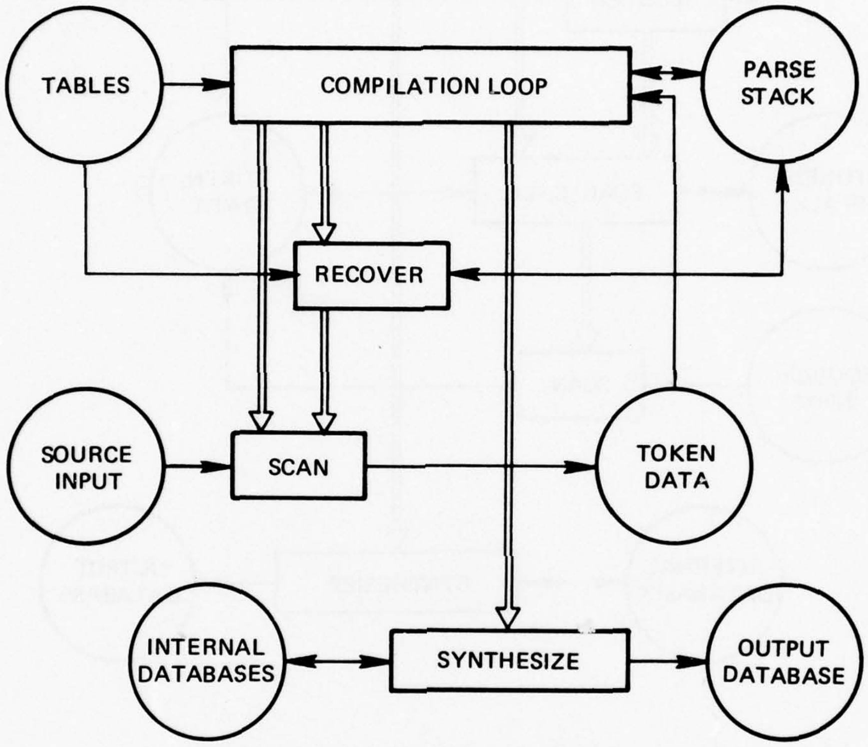


Figure 4-2. LALR(1) Compiler Structure

4.2.2.3 LALR(k) Extension

The DDDG's structure varies somewhat from the model described in the last section - this is primarily due to the structural complexity of the JOVIAL language.

Although the description of JOVIAL used to produce the DDDG's parsing tables is written in BNF, the resulting grammar is not LALR(1) (resolvable with a lookahead of one token), nor can it be straightforwardly reduced to LALR(1). This means that additional complexity must occur in the parsing algorithm to handle the increased lookahead.

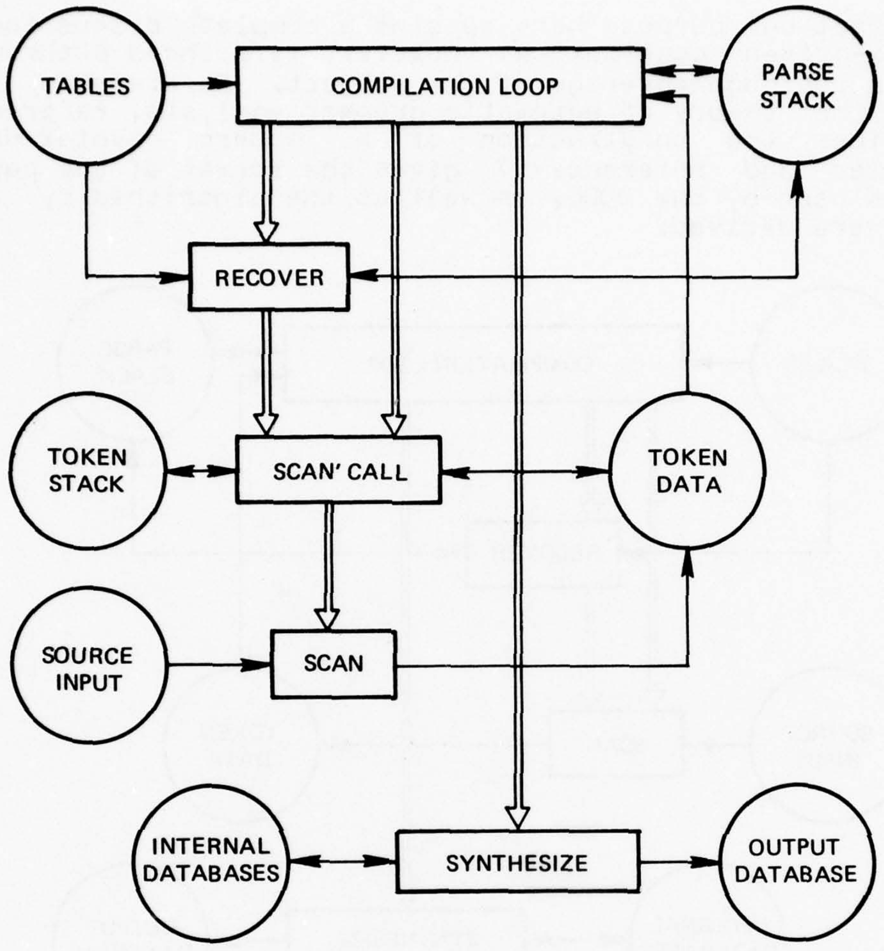


Figure 4-3. LALR(k) Compiler Structure

The difference in structure required by the addition of an LALR(k) parser is shown graphically in Figure 4-3. Notice the addition of the SCAN'CALL procedure, which controls the scanner. SCAN'CALL's primary job is to provide a buffer between the scanner and the parsing algorithm to preserve tokens that are passed over in "lookahead" situations. Normally, when it came time to read such tokens, they would have already disappeared. SCAN'CALL preserves them by stacking them in the token stack.

4.2.2.4 Modules Summary

This section describes the function and interaction of the abstract modules and databases pictured in Figure 4-4. Although Figure 4-4 presents an abstract view, the abstract procedure names are identical to actual procedure names in the DDDG, and some database names correspond to actual DDDG database names. This abstract view is important, because it defines away confusing detail which interferes with a coherent description. All description of DDDG operation will henceforth be based on Figure 4-4.

For illustrative purposes, Figure 4-4 is divided into two areas, as shown in Figure 4-5. Area I consists of processing routines; that is, routines and databases which are used in parsing and text-handling. Area II contains output routines used to generate the DDDG databases. Sections 4.2.2.4.1 and 4.2.2.4.2 describe the procedures and databases in Areas I and II, respectively.

4.2.2.4.1 Area I - Processing Routines

COMPILATION'LOOP - contains the parsing algorithm. Operates from [current token data], [parse stack], and [parsing tables] to determine whether the parse history as reflected in [parse stack] and [current token data] requires a reduction to a simpler syntactic form.

DATABASES

parsing tables - contain built-in information to guide LALR(k) parse

current token data - contains all associated information about the current token

parse stack - an internal stack-type database used to store parse history

OPERATION

When COMPILATION'LOOP determines that a reduction can be made to a simpler syntactic form, it makes the reduction, adjusts the parse history, and passes control to SYNTHESIZE. Otherwise, it may simply update the parse history and call for another token. It can call for another token in two ways - it can ask for a lookahead or a read token. In both cases, it calls SCAN'CALL with a

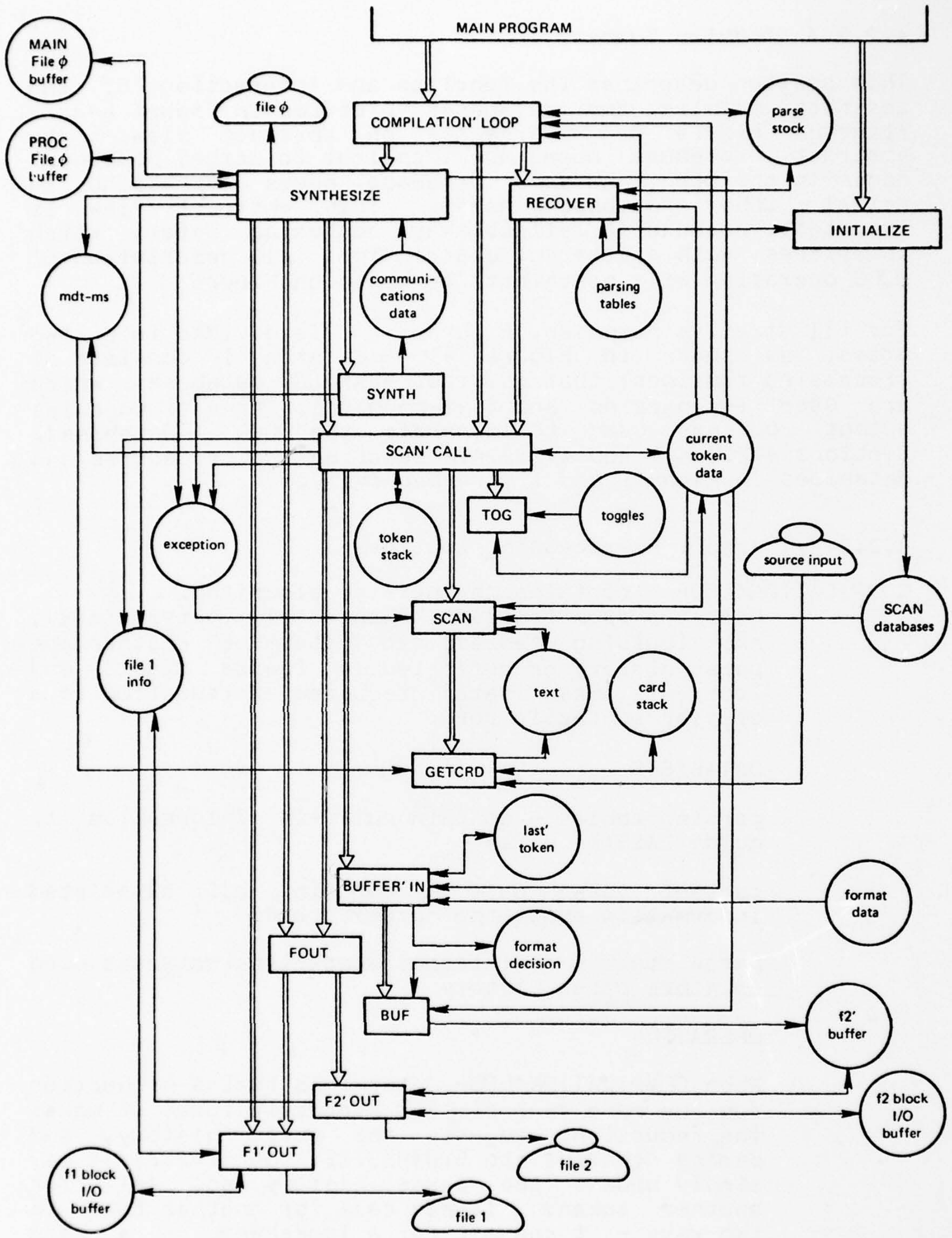


Figure 4-4. DDDG Structure

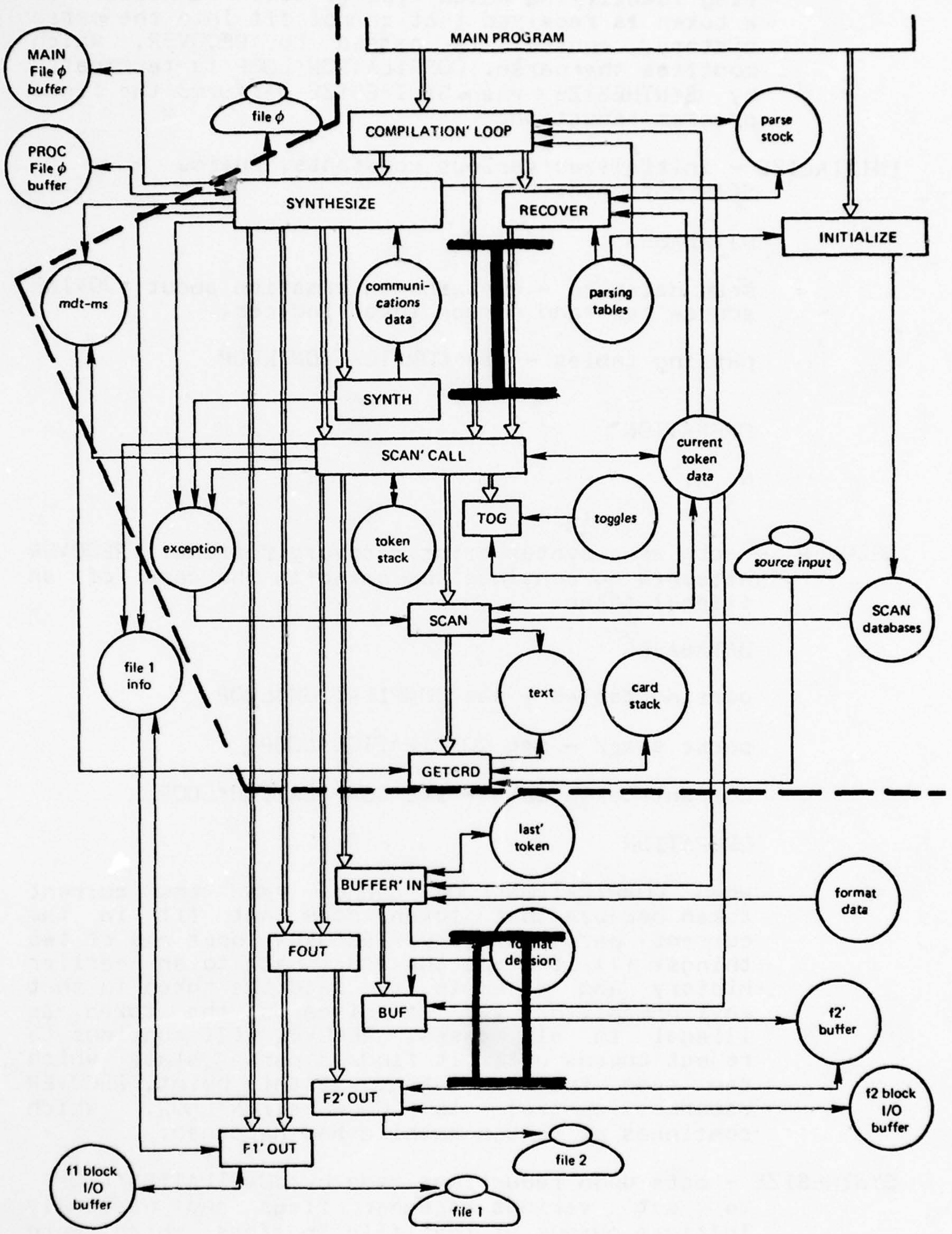


Figure 4-5. Partitioned DDDG Structure

flag identifying which type of token it needs. If a token is received that cannot fit into the parse history, control is passed to RECOVER, which modifies the parse. COMPILATION'LOOP is terminated by SYNTHESIZE when SYNTHESIZE performs the final program reduction.

INITIALIZE - initializes various constants, builds SCAN database.

DATABASES

scan database - contains information about JOVIAL source text and common token indices.

parsing tables - see COMPILATION'LOOP

OPERATION

N. A.

RECOVER - acts as a syntax error recovery routine. RECOVER attempts to continue the parse in the case of an illegal token.

DATABASES

parsing tables - see COMPILATION'LOOP

parse stack - see COMPILATION'LOOP

current token data - see COMPILATION'LOOP

OPERATION

When COMPILATION'LOOP cannot read the current token because that token does not fit in the current parse history, RECOVER does one of two things: (1) it wraps the stack back to an earlier history and attempts to read the token in that environment, or (2) it discards the token as illegal in all cases. RECOVER will continue to reject tokens until it finds a parse state which can read the next token. At this point, RECOVER returns control to COMPILATION'LOOP, which continues as though nothing had happened.

SYNTHESIZE - acts upon reductions made by COMPILATION'LOOP to set various scanner flags and basically initiate output of the file buffers which were packed by the output routines.

DATABASES

proc FILE 0 buffer - see 4.2.2.4.2, SYNTHESIZE.

main FILE 0 buffer - see 4.2.2.4.2, SYNTHESIZE.

current token data - see COMPILATION'LOOP

communications data - data passed from external procedure SYNTH which is essentially a part of SYNTHESIZE.

MDT-MS - macro stack and macro definition table

FILE 1 info - information about type of next FILE 1 record

FILE 0 - symbol table output file

exception - scanner exception flags

OPERATION

SYNTHESIZE recognizes certain types of key reductions, reductions to forms which it calls statement units. Each statement unit corresponds to a class of boxes in the flowchart output. Reduction to certain statement units causes SYNTHESIZE to call various file output routines. These reductions are processed by the SYNTH routine, which SYNTHESIZE calls when necessary. The most important Area I function performed by SYNTHESIZE is the setting of certain scanner flags and the related altering of the MDT. The scanner flags will be discussed in Section 4.3.2; however, the alteration of the MDT is an important high-level function. When a reduction is made that signals the reading of a new DEFINE directive, SYNTHESIZE searches the MDT for a name to match the DEFINE name. If it finds one, it uses that entry in the MDT as the new entry of the new definition. If it does not find one, SYNTHESIZE creates a new entry in the MDT and initializes it. In this way, SYNTHESIZE enables the the re-using of the DEFINE directive name as allowed in the JOCIT manual.

Clearly, SYNTHESIZE cannot fill in the macro entry in the MS. However, new space is always allocated in the MS for a new name, regardless of whether that name has been defined before (since the definition could now be longer than the last previous definition). SCAN'CALL eventually fills in the rest of the macro definition, after an appropriate scanner exception flag is set in a different SYNTHESIZE reduction case.

SCAN'CALL - primary function is maintenance of LALR(k) lookahead token stack. Serves also as comment processor, macro definition processor, and as output formatter, as well as causing the output of all comments.

DATABASES

current token data - see COMPILATION'LOOP

token stack - stack which contains previous lookahead tokens so that they can eventually be read by the parsing algorithm

MDT-MS - see SYNTHESIZE

exception - see SYNTHESIZE

OPERATION

If SCAN'CALL is called for a lookahead token, it calls SCAN for the next token only if it has looked at all the tokens in [token stack]. If the new token, or the token in token stack is part of a comment, it is saved, but ignored. If it is a macro name, it is saved and ignored, and SCAN is called again to expand the name. No output of information or processing of toggle comments or outputting of text can take place in lookahead situations, because the action would be temporally incorrect. The token may not actually be read until far in the future, and it would not do to have comments appearing where they did not occur, or toggles changing before the comment toggle.

When SCAN'CALL is called for a read token, it checks its [token stack] for available read tokens. If none exists, it calls SCAN. If the next token is the beginning of a comment, SCAN'CALL sets a special case flag for SCAN, and then collects the resultant tokens into a full comment and outputs it, with information on what type of comment it is (inline, stand-alone, etc.), to the output files. If the token is the beginning of a macro definition, SCAN'CALL fills in the appropriate entries in the MS with the text associated with the new macro name. Formatting decisions are made on whether to expand macros or not. Most formatting decisions are made in the buffer routines, however. Pre-defined macros are expanded as described above.

SCAN'CALL has a unique position in the structure of the DDDG - it is the only routine which has the power to alter scanner output or interpret it directly. Thus, SCAN'CALL is assigned various low-level tasks such as comment recognition and macro definition fill-in because to do these tasks elsewhere masks what is really happening - namely, a master-slave relationship between SCAN'CALL and SCAN.

TOG - decodes comment toggles.

DATABASES

current token data - see COMPILATION'LOOP

toggles - database listing all known toggle configurations

OPERATION

When called by SCAN'CALL, TOG checks the current line (which must be a comment of some kind) for occurrences of [<q><toggle>], where <toggle> belongs to the set of recognizable toggles in [toggles]. Some of the legal toggles include ASIS, DEBUG, and EXPAND. The presence of or lack of <q>, where <q> represents the character "'", indicates the desire to either turn off or turn on the toggle.

SCAN - decodes current line and returns next legal token, along with various related information.

DATABASES

current token data - see SCAN'CALL

text - the current line of text, as presented by GETCRD

scan database - contains information about JOVIAL source text and common token indices.

exception - see SYNTHESIZE

OPERATION

The scanner operates in two modes - special, and regular. Regular mode means that the scanner uses its databases to branch to an appropriate routine based on the first character of the next token. It then, in these special cases, decodes the token further and places it and its associated information into [current token data]. The associated information includes the number of blanks preceding the token, the token's actual character appearance, whether the token is a macro name or part of a macro expansion (DEFINE directive expansion), and whether a carriage-return line-feed immediately precedes the token. This extraneous information is needed to enable the EXPAND and ASIS options.

In special mode, the scanner is doing something unusual because of the unusual JOVIAL construction it is processing. Usually, this involves returning a string of characters (as in a comment or define directive) rather than dividing the line up into real tokens. At other times special mode can involve returning specially designated tokens because of the scanner's inability to differentiate between structurally similar constructs. This special mode is enabled by flag-setting in the SYNTHESIZE and SYNTHESIZE routines, since only the parsing algorithm knows what construct is currently being processed, and it must decide that a special scanner mode is needed.

GETCRD - places the next input card into [text].

DATABASES

text - see SCAN

source input - input to flowcharter

MDT-MS - see SYNTHESIZE

card stack - used to stack cards holding multiple card macro definitions, or multiply nested macro definitions

OPERATION

In most cases, GETCRD behaves as one might expect - namely, reading in the next card from the source file and placing it in [text]. When a macro name has just been read by SCAN, however, GETCRD pulls the definition of the macro from MS, and loads it into [card stack]. On succeeding calls to GETCRD, the routine reads from [card stack] until it is empty. A nested macro call thus requires no special GETCRD mechanism - its definition, too, is loaded on top of the already occupied [card stack]. Obviously, recursive macro calls will eventually overflow the card stack, which is pointed out to the user with an appropriate overflow message.

4.2.2.4.2 Area II - Output Routines

SYNTHESIZE - sets flags which trigger output; performs all FILE 0 output.

DATABASES

proc FILE 0 buffer - contains names of all procedures and functions called within scope of current proc.

main FILE 0 buffer - contains names of all procedures and functions called within main procedure proper.

(for other data items used, see 4.2.2.4.1, SYNTHESIZE)

OPERATION

When SYNTHESIZE detects the occurrence of a statement unit reduction, it calls FOUT, which outputs the appropriate data to files 1 and 2. [FILE 1 info] tells FOUT what type of FILE 1 record to build. In the case of certain "dummy" FILE 1 entries (end of scope), SYNTHESIZE calls FILE1'OUT directly.

When a procedure or function call is detected in SYNTHESIZE, the current scope is checked. If the parse is within an internal procedure, the new proc or function name is checked against the contents of [proc FILE 0 buffer]. Otherwise, it is checked against [main FILE 0 buffer].

When a proc scope terminates, [proc FILE 0 buffer] is dumped out to FILE 0. When a new proc scope is entered, the name of the proc is entered into [proc FILE 0 buffer] and the old contents of the buffer are flushed. [Main FILE 0 buffer] is dumped at the end of the parse.

If it overflows, [proc FILE 0 buffer] can be dumped, and a new buffer is then built. An overflow of [main FILE 0 buffer] causes a non-fatal error message and another message which eventually appears on the Invocation Diagram produced by the DDG.

BUFFER'IN - makes formatting decisions for the next FILE 2 record.

DATABASES

last'token - number of last token entered into f2'buffer

format data - decision tables for formatting of FILE 2 output

current token data - see 4.2.2.4.1, COMPILATION'LOOP

format decision - variables set by BUFFER'IN which communicate formatting decisions to BUF.

OPERATION

BUFFER'IN simply queries [format data], which contains enough information to enable the format decisions for the next FILE 2 entry.

BUF - adds to [f2'buffer] the formatted character representation of the current token.

DATABASES

format decision - see BUFFER'IN

current token data - see 4.2.2.4.1, COMPILATION'LOOP f2'buffer - holds the character representation of the current statement unit

OPERATION

BUF appends to [f2'buffer] a formatted version of the current token's character representation. Its primary function is adding or deleting spaces from before or after the token. BUF also keeps a byte and line count of the contents of [f2'buffer]. For illustrative purposes, this information is assumed to be part of [f2'buffer].

F2'OUT - writes out one FILE 2 record.

DATABASES

FILE 1 info - see 4.2.2.4.1, SYNTHESIZE

f2'buffer - see BUF

FILE 2

f2'block I/O buffer - buffer used to block FILE 2 records together

OPERATION

F2'OUT writes the contents of [f2'buffer] out to [f2'block I/O buffer]. It also copies the byte count and line information from [f2'buffer] into [FILE 1 info].

If [f2'block I/O buffer] fills up, the buffer is output as a FILE 2 record. The buffer is always output at the end of the DDDG, to account for partial filling followed by DDDG termination

F2'OUT always flushes f2'buffer.

F1'OUT - writes out one FILE 1 record.

DATABASES

FILE 1 info - see 4.2.2.4.1, SYNTHESIZE

f1'block I/O buffer - buffer used to block FILE 1 records together

FILE 1

OPERATION

From information in [FILE 1 info], F1'OUT puts one FILE 1 record out to [f1'block I/O buffer]. See F2'OUT for a description of how this buffer is used.

4.2.2.5 Operation

Having described the purposes of the modules and databases pictured in Figure 4-4, it remains to be shown how the system works as a whole. The most straightforward way of accomplishing this is to trace the execution of a test example using the descriptions in Section 4.2.2.4.

The test example chosen (see Figure 4-6) shows DDDG formatting capability, syntax error recovery, the ASIS option, FILE 0 construction, macro expansion, toggle processing, and comment handling, as well as the usual FILE 1 and FILE 2 construction. An IFEITH construct is included for statement unit variety and END'SCOPE illustration purposes. Note that this program will not compile successfully.

Execution of the DDDG begins with INITIALIZE, which builds the scanner database and presets certain internal variables. Then control is passed to COMPILATION'LOOP (C'LOOP), which will direct the DDDG for the remainder of its execution.

C'LOOP calls SCAN'CALL for a read token in the input text. SCAN'CALL calls SCAN, which calls GETCRD. GETCRD returns a line of text (START $), and SCAN identifies the first token in the line as "START". SCAN'CALL then calls BUFFER'IN, BUFFER'IN calls BUF with its formatting decision, and BUF puts out "START " to [f2'buffer]. SCAN'CALL returns, and COMPILATION'LOOP interrogates [parsing tables] and [current token data] to determine a proper parsing state transition. No reduction is made, so C'LOOP calls SCAN'CALL for another token. "$" is returned in the same fashion as "START", and C'LOOP performs a reduction of "START $" to <PROGRAM HEAD>.

C'LOOP then calls SYNTHESIZE, which fills [FILE 1 info] with statement unit number 39, indicating the recognition of a <program head>. SYNTHESIZE calls FOUT, which calls F2'OUT. F2'OUT writes the byte and line count of [f2'buffer] to [FILE 1 info], moves [f2'buffer] to [f2'block I/O buffer], and clears [f2'buffer]. F2'OUT returns, and FOUT calls F1'OUT, which outputs an [f1'block I/O buffer] record based on information in [FILE 1 info]. Control returns to C'LOOP.

```
start$
'' [expand] ''
         ''[debug]   ''
define integer ''i 36                s''$

item         aa            integer  $

''this next stmt will cause a parse error''
''['debug]''

aa=ifeith bb$
aa=3$
ifeith aa eq 3$ begin
          bb=1$
''          [debug]''
          print$
end



     ''['debug]''

orif 1$ begin
          aa=1$
end

end ''of ifeith''
''now, let's turn on asis [asis]''

print$

proc print$ begin

         print'it                   $
end
''    ['asis]  ''
term$
```

Fiqure 4-6. Example JOVIAL J3 Program

C'LOOP calls SCAN'CALL for a read token, SCAN'CALL calls SCAN, SCAN calls GETCRD, and returns the token "//". But SCAN'CALL knows that it is not processing a DEFINE directive, so it must have a comment. It knows the comment belongs on its own line, also, because the CRLF flag was set by GETCRD. SCAN'CALL calls TOG to process the comment for toggles. TOG finds "[EXPAND]", and sets the EXPAND toggle flag. SCAN'CALL then acts as SYNTHESIZE did, on the previous reduction - namely, writing a "38" into [FILE 1 info], and calling FOUT. SCAN'CALL then loops for a real token, and the same activity occurs with the new card, " [DEBUG] ".

Finally SCAN returns a real token, "DEFINE". The parse can be picked up at this point in Figure 4-7 (thanks to the [DEBUG] toggle). Nothing that has not already been described occurs until reduction 17 is performed. SYNTHESIZE enters the DEFINE name into the [MS], and the define and scanner exception flags are set. SCAN'CALL is called for the next token, and calls SCAN. SCAN returns "<characters>", because of the exception flag. Interrogating the define flag, SCAN'CALL places the character representation of <characters> into the [MDT] and updates the [MS] entry of the current macro. The next token returned is "//", so the flags are turned off and the parse continues.

Operation continues normally, until SCAN encounters the string "integer" in the next card. SCAN correctly identifies "integer" as an identifier, but before passing <identifier> to SCAN'CALL, it searches the [MS] for "integer". It finds it, and calls GETCRD with a flag set. GETCRD copies the definition of "integer" from the [MDT] to [card stack], after stacking what remains of the current card ("$"). GETCRD then returns the macro definition as the current text string. Before the call to GETCRD, however, a return was made to SCAN'CALL. During this return processing, SCAN'CALL interrogated the EXPAND toggle flag, decided to expand the macro, discarded the macro name, and re-invoked SCAN. SCAN had previously signalled itself that it needed a new card by moving its current card pointer past the text limit on the card, which is why it called GETCRD.

When GETCRD returns the macro definition as the current card, the tokens on the card are passed back to SCAN'CALL in the usual fashion by SCAN, except that the tokens are flagged as part of a macro definition. Interrogating the EXPAND toggle, SCAN'CALL writes these tokens out to [f2'buffer]. When GETCRD is called for another card (the macro definition having been exhausted), it returns what is left of the original card ("$"). The parse then continues normally.

```
    READ TOKEN RETURNED: DEFINE
    READ TOKEN RETURNED: <IDENTIFIER>
    #16   <DEFINE HEAD> ::= DEFINE <IDENTIFIER>
    READ TOKEN RETURNED: ''
    #17   <''> ::= ''
    READ TOKEN RETURNED: <CHARACTERS>
    #18   <TEXT> ::= <CHARACTERS>
    READ TOKEN RETURNED: ''
    READ TOKEN RETURNED: $
    #15   <DEFINE DIRECTIVE> ::= <DEFINE HEAD> <''>  <TEXT>
    '' $
    #13   <DIRECTIVE> ::= <DEFINE DIRECTIVE>
    #12   <ELEMENT> ::= <DIRECTIVE>
    #8   <ELEMENT LIST> ::= <ELEMENT>
    READ TOKEN RETURNED: ITEM
    READ TOKEN RETURNED: <IDENTIFIER>
    READ TOKEN RETURNED: I
    READ TOKEN RETURNED: <NUMBER>
    READ TOKEN RETURNED: S
    #212   <SIGNING> ::= S
    #209   <INT HEAD> ::= I <NUMBER> <SIGNING>
    LOOKAHEAD TOKEN RETURNED: $
    #205   <INTEGER DESCRIPTION> ::= <INT HEAD>
    #199   <ITEM DESCRIPTION> ::= <INTEGER DESCRIPTION>
    READ TOKEN RETURNED: $
    #196    <SIMPLE ITEM DECLARATION> ::= ITEM <IDENTIFIER>
    <ITEM DESCRIPTION> $
    #178   <DATA DECLARATION> ::= <SIMPLE ITEM DECLARATION>
    #30   <DECLARATION> ::= <DATA DECLARATION>
    #11   <ELEMENT> ::= <DECLARATION>
    #9   <ELEMENT LIST> ::= <ELEMENT LIST> <ELEMENT>
---------------------- point1
    ILLEGAL SYMBOL PAIR: =  IFEITH
    PARTIAL  PARSE  TO  THIS  POINT  IS:   <PROGRAM   HEAD>
    <ELEMENT LIST> <VARIABLE> =
    SKIPPED OVER TOKEN "IFEITH"
    RESUMING
---------------------- point2
    READ TOKEN RETURNED: <IDENTIFIER>
    LOOKAHEAD TOKEN RETURNED: $
    #169   <PROC NAME> ::= <IDENTIFIER>
    READ TOKEN RETURNED: $
    #167   <PROCEDURE CALL> ::= <PROC NAME> $
    #141   <SIMPLE STATEMENT> ::= <PROCEDURE CALL>
    #22   <STATEMENT> ::= <SIMPLE STATEMENT>
    #10   <ELEMENT> ::= <STATEMENT>
    #9   <ELEMENT LIST> ::= <ELEMENT LIST> <ELEMENT>
    READ TOKEN RETURNED: END
    #397   <END> ::= END
    #77   <COMPOUND  STATEMENT> ::= <BEGIN> <ELEMENT LIST>
    <END>
    #89   <THEN CLAUSE> ::= <COMPOUND STATEMENT>
```

Figure 4-7. Partial Parse of Example Program

Little of note occurs (except for the [DEBUG] toggle being turned off at card 10) until the presence of a parse error in card 12. The output between points 1 and 2 in Figure 4-7 shows RECOVER's response to the illegal IFEITH token. It could not wrap the state stack back to a point where IFEITH was a legal state transition symbol, so it rejected the token.

Card 17 is a procedure call - SYNTHESIZE detects this, and enters the name "print" into [MAINP FILE 0 buffer]. Card 18 causes a variation from the usual output of both a FILE 1 and FILE 2 record - after the ordinary call to FOUT, SYNTHESIZE makes an additional call to F1'OUT, which causes a dummy FILE 1 record to be produced, called an END'SCOPE (identified in Figure 4.2-7 by a STMT'TOKEN type of "1"). END'SCOPEs are used by the DDG to map out the flowchart intelligently.

Card 29 turns on the [ASIS] toggle - notice the effect on files 1 and 2 (Figure 4-8). Parsing continues, but output is totally controlled by the CRLF flag. FILE 1 records which are ASIS are identified by "37", and are printed in one contiguous box by the DDG. Leading spaces are preserved in the FILE 2 text by using the BLANKS information in [current token data] to concatenate the appropriate spaces.

Card 33 causes the initialization of [proc FILE 0 buffer] by SYNTHESIZE, SYNTHESIZE having recognized a reduction to <procedure call>. SYNTHESIZE also sets the scope to 1, which means that proc and function calls following will be recorded in [proc FILE 0 buffer] instead of [mainp FILE 0 buffer]. Thus, card 35's procedure name is entered into [proc FILE 0 buffer]. Card 36 causes SYNTHESIZE to recognize a reduction to <procedure declaration>, which resets the scope to 0. Figure 4-9 shows the final FILE 0 output.

At termination, all temporary I/O buffers are written out. Notice card 37, which turns off the [ASIS] toggle. It is necessary to follow the User's Manual rules about the occurrence of [ASIS] and ['ASIS], since improper usage will cause the DDG to miss vital END'SCOPE information, and will result in no flowchart being produced.

FILE 1:

INDEX	TOKEN	RECS	LNGTH
0	39	1	8
1	38	1	15
2	38	1	15
3	9	1	43
4	25	1	17
5	38	1	44
6	38	1	13
7	18	1	17
8	18	1	9
9	49	1	7
10	48	1	10
11	4	1	6
12	18	1	9
13	38	1	22
14	22	1	8
15	5	1	4
15	1	1	4
16	38	1	13
17	46	1	9
18	4	1	6
19	18	1	9
20	5	1	4
20	1	1	4
21	5	1	4
21	1	1	4
22	3	1	14
23	37	1	35
24	37	1	6
25	37	1	17
26	37	1	54
27	37	1	3
28	38	1	18
29	8	1	7
29	1	1	7

Figure 4-8. FILE 1 and FILE 2 of Example Program

FILE 2:

```
 0 START $
 1 '' [EXPAND] ''
 2 ''[DEBUG]    ''
 3 DEFINE INTEGER ''I 36                   S'' $
 4 ITEM AA I 36 S $
 5 ''THIS NEXT STMT WILL CAUSE A PARSE ERROR''
 6 ''['DEBUG]''
 7 AA = IFEITH BB $
 8 AA = 3 $
 9 IFEITH
10 AA EQ 3 $
11 BEGIN
12 BB = 1 $
13 ''            [DEBUG]''
14 PRINT $
15 END
16 ''['DEBUG]''
17 ORIF 1 $
18 BEGIN
19 AA = 1 $
20 END
21 END
22 ''OF IFEITH''
23 '' NOW, LET'S TURN ON ASIS [ASIS]''
24 PRINT$
25 PROC PRINT$ BEGIN
26         PRINT'IT                    $
27 END
28 ''    ['ASIS]  ''
29 TERM $
```

Figure 4-8. FILE 1 and FILE 2 of Example Program (cont.)

FILE 0:

```
***
PRINT                  -- procedure name
PRINT'IT               -- name of procedure called
------------------------------
***
MAIN                   -- main program
PRINT                  -- name of proc called
------------------------------
```

Figure 4-9. FILE 0 of Example Program

4.3 Phase 1 Modules, Variables, and Constants

This section contains data items and lower level procedure descriptions. It is broken up as follows: Section 4.3.1 covers variables and data structures; Section 4.3.2 contains procedure descriptions; and Section 4.3.3 contains local declarations.

4.3.1 Phase 1 Variables and Data Structures

Section 4.2.2.4 gives an abstract view of the functions of various data within the DDDG. This section explicitly lists data items by name, as they are used in the program. It will be difficult for the reader to draw parallels between this list and the abstractions in Section 4.2.2.4 -- this list is intended only as a reference for quick lookup of variable names. The listing of the phase 1 data compool is organized in the same way as the abstraction in 4.2.2.4 -- thus, parallels between the two are easily drawn.

This section is divided as follows: Section 4.3.1.1 contains DEFINE directives; Section 4.3.1.2 contains table declarations; Section 4.3.1.3 contains file declarations; Section 4.3.1.4 contains global variable declarations; and Section 4.3.1.5 contains declarations for the parsing tables.

4.3.1.1 Phase 1 DEFINE Directives

```
define asis'stmt ''37''$

define asis ''0''$

define character ''h 150'' $

define debug ''2''$

define expand ''1''$

define f1'blocksize ''78''$

define f0'blksize ''60''$

define f2'blocksize ''12''$

define false ''0'' $

define integer ''i 36 s'' $
define macro ''1000''$

define max'msp ''25''$

define max'symbuf ''1000''$

define max'mtbl ''50''$

define max ''50''$ ''token stack size''

define max'f2 ''50''$

define max'mainp ''1000''$

define max'mdt ''100''$

define true ''1'' $

define type3 ''37''$

define type1 ''3''$

define type2 ''2''$
```

4.3.1.2 Table Declarations (Output Record Formats)

```
table file2b r f2'blocksize  s  n$  ''fixed  lngth,  serial,
nopacking'' ''each entry in this table is a file2 record''
 begin
 item f2'entry h 150$
 end

table  file1b r f1'blocksize s n$ ''each entry in this table
is a file1 record''
 begin
 item file2'index i 36 s$ ''index into file2''
 item stmt'token i 36 s$ ''stmt type''
 item f2'recs i 36 s$ ''no of file2 records''
 item stmt'lngth i 36 s$ ''total no of bytes''
 end

table file0b r f0'blksize s n $ begin ''each output of  this
table is a file0 record''
 item f0'entry h 30$ end
```

4.3.1.3 File Declarations

file source h 12000 v 80 v(a) v(b) v(c) v(d) v(eof) 11 $ ''file declaration of source file''

file quab h 12000 v 118 12$ ''used for error output file''

file file1 b 10000 r 313 13$

file file2 b 10000 r 301 14$

file file0 b 10000 r 301 15$

''these are the declarations for the three phase1 output files''

4.3.1.4 Global Variables and Data Structures

item blanks integer p 0$ ''contains the number of blanks before the current token (counting from the last token, or from the beginning of the current card)''

item buf'lines integer p 1$ ''number of lines in current statement unit''

item bcd character $ ''character representation of current token''

item buffer h 80 $ ''used to input one 80-column source card''

item comment i 36 s p 0$ ''contains the type of the current comment - i.e., whether it is within the scope of the preceding stmt, on the same line as the preceding statement, or on its own line ''

item comment'flag integer p 0$ ''used to strip spaces from in front of poorly constructed comment lines''

item crlf integer p 0$ ''if crlf is 1, a carriage-return line-feed has occurred before the current token''

item char'count integer p 0$ ''number of characters in current statement unit''

item card'count integer $ ''number of cards read''

item cp integer p 1 $ ''points at next available character in text''

```
item compiling b$ ''switch to turn compilation on and  off''

item  context  integer  p  0$  ''identifies whether parse is
within a procedure definition or in the main program''

item characters integer $ ''index of <characters> in VOCAB''

array callcount 5 integer $ begin 0 end  ''used  to  analyze
control flow in phase 1''

item define'flag integer p 0$ ''flags the fact that a DEFINE
directive is being processed''

item digit integer p 8 $ ''identifies digits in array TYPE''

item dollar integer $ ''index of $ in VOCAB''

item dot integer $ ''index of . in VOCAB''

item ddone b$ ''binary flag for while loops''

item ddcount integer p 0$ ''utility variable''

item divide integer $ ''index of / in VOCAB''

item ellipsis integer $ ''index of ... in VOCAB''

item equal integer $ ''index of = in VOCAB''

item exchng integer $ ''index of == in VOCAB''

item eofile integer $ ''index of end-of-file in VOCAB''

item exp integer $ ''index of ** in VOCAB''

item  exception  integer  p 0 $ ''contains scanner exception
case number''

item f2'entries integer p 0$ ''number of file2 entries''

item f2bindex integer p 0$ ''index of last file2 entry''

item f2bbytes integer p 0$ ''no of bytes in last f2  entry''

item  f0'blocksize  integer  p  60$  ''size  of a file0 disk
block''

item f1'entries integer p 0$ ''number of file1 entries''

item f2brecs integer p 0$ ''no of records in last f2 entry''
```

array f2'buffer max'f2 character$ ''buffer contains statement unit text to be output to file2''

array fixl 100 i 36 s$ ''not used''

array fixv 100 i 36 s$ ''not used''

item ident integer $ ''index of <identifier> in VOCAB''

item idloop integer p 0$ ''utility variable''

item iddone integer p 0$ ''utility variable''

item kluge'flag integer p 0$ ''not used''

item kk i 36 s p 1$ ''utility variable''

item kkk integer$ ''not used''

item last'ident character$ ''contains the character representation of the last identifier encountered by SCAN''

item last'token integer$ ''contains the index of the last token placed into f2'buffer by BUFFER'IN''

item left'abs integer $ ''index of (/ in VOCAB''

item left'paren integer $ ''index of (in VOCAB''

item letter integer p 9 $ ''identifies letters in array TYPE''

item left'exp integer $ ''index of (* in VOCAB''

item left'subs integer $ ''index of ($ in VOCAB''

array lengths 30 integer $ ''contains lengths of various terminal symbols''

item macro'flag integer p 0$ ''flags the fact that the current token is part of a macro expansion''

item macro'name integer p 0$ ''index into macro tables of current macro name''

item ms'flag integer p 0$ ''used by GETCRD to remember that it is in the process of expanding a macro''

item max'nospab integer p 7$ ''size of nospab''

item max'nospaa integer p 4$ ''size of nospaa''

```
item max'nopair integer p 18$ ''size of npair1 and nopair2''

item  msp  integer  p  24$ ''points at next free location in
ms.''

item max'macro integer p 0$ ''points at next  free  location
in macro table (MS)''

item mant'e integer $ ''index of e in VOCAB''

item  mp i 36 s$ ''points at left edge of reduction phrase''

item  mainp'ptr  integer  p  2$  ''points   to   next   free
mainp'calls location''

item  mptr  integer  p  0$ ''internal variable to save macro
entry position''

item mantissa integer $ ''index of <mantissa> in VOCAB''

item mpp1 i 36 s$ ''mp+1''

array ms max'msp character$ ''stack used  for  GETCRD  macro
definition expansion''

array mname max'mtbl character$ ''mname, mstart, mlength are
parallel  arrays  which make up the macro table (MS).  mname
is the name of the macro''

array mlength max'mtbl integer$ ''length  of  definition  in
MDT''

array  mdt  max'mdt  character$  ''contains  text  of  macro
definitions''

array mainp'calls max'mainp h 30$ ''file0 array to save main
program  procedure  calls,  initialized  to  indicate  main
program calls'' begin 3h(***) 4h(maIN) end

array  mstart  max'mtbl  integer$  ''index  of  beginning of
definition in MDT ''

item nil character $ ''initialized in initialize to  nil  ''
''contains 150h([00000     ...      )''

item  next'mfree integer p 0$ ''points at next free location
in MDT''

item number integer $ ''index of <number> in VOCAB''

item number'value integer $ ''contains value of number  read
in''
array  nospab  8  integer$  begin 37 40 38 39 6 55 105 9 end
''list of tokens which never have spaces before them''

array  nopair1  19  integer  $  begin  103  103  103  18  32
''nopair1(  i  ) and nopair2( i ) cannot have spaces between
them'' 30 11 106 15 11 67 40 71 73 64 63 84 60 25 end

array nospaa 5 integer$ begin 104 55 105 2 37 end ''list  of
tokens which never have spaces after them''

array nopair2 19 integer$ begin 18 30 11 2 2 2 2 2 103 103 2
2 2 2 2 2 2 2 103 end

item  outtok  integer$  ''signals  SYNTH that a file1 record
should be built, with stmt unit = outtok''

item outscope integer$ ''signals SYNTH that a file1 endscope
record should be built''

item phony integer p 0$ ''tells GETCRD to  expand  the  most
previous  macro  name,  and  to pass as the next source card
that expansion''

item prime integer $ ''index of single quote in VOCAB''

item quote integer $ ''index of two single quotes in VOCAB''

item real'macro'flag integer p  0$  ''usually  the  same  as
macro'flag,  except  in lookahead situations or when reading
former lookahead tokens from token stack''

item reserved'limit integer $ ''points at limit of  reserved
words in VOCAB''

item right'exp integer $ ''index of *) in VOCAB''

item right'abs integer $ ''index of /) in VOCAB''

item right'subs integer $ ''index of $) in VOCAB''

item scale'a integer $ ''index of a in VOCAB''

item star integer $ ''index of * in VOCAB''

item stacksize i 36 s p 100$ ''size of state stack''

item save'tog integer$ ''used by synthesize to set and reset
asis''

item  sp  i  36 s$ ''state stack pointer - points at current
state''
```

```
item symbuf'ptr integer p 0$ ''points to  next  free  symbuf
location''

item state i 36 s$ ''contains current state name''

array  state'stack 100 i 36 s$ ''used by COMPILATION'LOOP to
save state numbers in LALR(k) parse''

array  symbuf  max'symbuf  h  30$  ''file0  array  to   save
procedure calls within procedures''

item togmax integer p 2$ ''size of tog and togc''

item  tsmax  integer  p  0$  ''points  to mext free entry in
stack''

item tsbegin integer p 0$ ''points to  next  read  token  in
stack''

item  text  character  $  ''contains  current text line from
GETCRD''

item token integer $ ''contains index of  current  token  in
VOCAB''

item tempmk character$ ''temporary''

item temph2 character$ ''temporary''

item tempi2 integer$ ''temporary''

item temph3 character$ ''temporary''

item tbegin integer$ ''index of begin in VOCAB''

item tend integer$ ''index of end in VOCAB''

item tempcl h 150$ ''temporary''

item  total'f2'entries  integer  p  0$  ''total  no of file2
records''

item tempid h 30$ ''temporary''

item tsptr integer p 0$ ''points to next lookahead token  in
token stack''

item  text'limit  integer  p  0  $  ''contains index of last
character in current text line''
```

item temphl character$ ''temporary''

item tempi3 integer$ ''temporary''

item tbeginl integer$ ''index of beginl in VOCAB''

item temph4 character$ ''temporary''

item tempil integer$ ''temporary''

item tempc h 150$ ''tempc and following two variables are temporaries''

item tv integer$ ''index of v in VOCAB''

array togc 10 h 6$ begin 6h(asis) 6h(expand) 6h(debug) end ''contains legal toggle names''

array tsn max integer$ ''tsn,tsc,tsbl,tscrlf,tsmflag are parallel arrays called the token stack. tsn is the number of the token - same as token''

array tsbl max integer$ ''same as blanks - refers to tsn(i), though.''

array tsmflag max integer$ ''same as macro'flag - refers to tsn(i).''

array type 64 integer $ begin 10 end ''returns type of ascii character, where value of character is index into type''

array tog 10 integer$ begin 0 0 0 0 0 0 0 0 0 0 end ''contains toggle values''

array tsc max character$ ''same as bcd - refers to tsn(i)''

array tscrlf max integer$ ''same as crlf - refers to tsn(i), though.''

4.3.1.5 Declarations for Parsing Tables

array apply1 447 i 16 1 $ ''matches states on state stack for apply state transitions''

array apply2 447 i 16 s $ ''contains transitions corresponding to matches in apply1''

item asize i 16 s p 446 $ ''size of apply arrays''

```
array  index1    856  i 16 s $ ''points into read1 and look1
arrays; contains push states''

array index2   856  i 16 s $ ''contains count  correspondin
to index1-identified state''

array  look1    170  i 16 s $ ''matches symbols in lookahead
states''

array  look2    170   i   16   s   $    ''contains   transitions
corresponding to matches in look1''

item lsize i 16 s p 169 $ ''size of look arrays''

item  maxln  i  16 s p 456 $ ''start of push states in index
arrays''

item maxpn i 16 s p 456 $ ''start of apply states  in  index
arrays''

item  maxrn  i  16  s p 395 $ ''start of lookahead states in
index arrays''

item maxsn i 16 s p 855 $ ''largest state number''

array  nproduce'name    400   i  16   s   $   ''used  to  print
productions''

array   nstate'name    400   i  16   s   $   ''used  to  print
productions''

item pn i 16 s p 399 $ ''number of productions''

array read1   4147  i 16   s   $   ''matches  symbols  in  read
states''

array   read2    4147   i  16   s   $   ''contains  transitions

corresponding to matches in read1''

item rsize i 16 s p 4146 $ ''size of read arrays''

item start'state i 16 s p 1 $ ''state to start parse in''

array  state'name    396   i  16   s   $    ''used   to   print
productions''

item terminaln i 16 s p 106 $ ''number of terminal symbols''

array  vocab   285  h  30  $  ''vocabulary of all symbols in
grammar''

item vocabn i 16 s p 284 $ ''size of vocab''
```

4.3.2 Procedure Descriptions

The following is a list of DDDG procedures and accompanying descriptions of their function/operation. These procedures are listed in the order in which they appear in the DDDG source listing.

proc file1'out(stmt'type)$

> Performs all file1 output. Stmt'type is the type of statement unit that the next file1 record will contain.
>
> File1'out checks to see if table file1b is full; if so, it outputs the table and zeroes f1'entries. It then sets the next free row of table entries to the correct statement unit type, and loads the correct file2 information from variables set by file2. F1'entries is then incremented.

proc fout(nerd) $

> Calls file1'out and file2'out if the asis toggle is off. Nerd contains the statement unit number to be passed to file1'out.

proc file2'out$

> Performs all file2 output.
>
> Operation is same as file1'out, except that the number of lines, the number of characters, and the current absolute record number of file2 are saved for use by file1'out.

proc out(aa,cc)$

> See description of string package in Section 4.1

proc substr(aa,first,num)$

> See description of string package in Section 4.1

proc cat(aa,bb)$

> See description of string package in Section 4.1

proc cnvert(aa)$

> See description of string package in Section 4.1

proc spaces(num)$

See description of string package in Section 4.1

proc null(aa)$

See description of string package in Section 4.1

proc length(aa)$

See description of string package in Section 4.1

proc recover$

Performs syntax error recovery functions. See Section 4.2.2.4.1 for a good high-level description of this routine's function.

Recover begins by looping back through state'stack until sp = 0. If on the way one of those states can read the current (illegal) token (this is determined by proc noconflict), the parse is re-started in that state. Otherwise, the token is rejected and scan'call is called for a new token.

proc noconflict(current'state) $

Searches current'state's read1 array to find a match to the current token. If there is one, noconflict returns 1. If not, it returns 0.

proc synthesize(production'number)$

Controls the creation of the DDDG output databases (files 0-2). Production'number is passed by compilation'loop parsing algorithm.

Synthesize incorporates code associated with several important syntactic reductions made by the parser. All other reduction code is in proc synth, which is just an external continuation of synthesize.

Reduction to <program>: Turns off compilation flag "compiling." Writes out mainp'calls array, which contains procs called by the main program. If mainp'ptr is > than the file0 blocksize (f0'blocksize), then the array is written out to more than one file0 record.

Reduction <''> -> '': Sets exception=1 for scan. Then uses last'ident to enter macro name into macro table (mname). If the name is already defined, the old mname, mstart, and mlength entries are used; if not, a new entry is created. M'length and mstart are initialized to 0 and next'mfree (a pointer to next free mdt entry) respectively.

Reduction to first part of an array declaration: Calls scan'call for tokens until it determines that the array declaration is initialized or non-initialized. If it is initialized, the begin tokens read are changed to begin1s. This code makes for a significant reduction in parsing table size and complexity.

Reduction to procedure or function call: If context=1, the name of the proc is looked up in symbuf array - if it is there, it is ignored; if not, it is appended to symbuf. If the context=0, the same is done with the mainp'calls array. If either of these arrays overflows, a message appears on file0 and on the error file. Context is set in synth cases marking the beginning/end of a procedure/function.

If production'number is not equal to one of the above reductions, synth is called. On return, if synth has set outscope, an endscope file1 record is written. If outtok was set, file1 and file2 records are written with stmt type outtok. An examination of the synth source listing should identify where these variables are set.

proc stack'dump$

Called during a parse error to dump the names of the states in state'stack.

proc print'production(prodn,left'stackn,right'stackn)$

Called by synthesize when the debug toggle is set. Prints the BNF production which has just been applied in the parse. Prodn is the production number, left'stackn points at the new state number, and right'stackn points at the rightmost state in that part of state'stack involved in the production.

proc err(aa,bb)$

Puts out illegal characters encountered by scan to error file.

proc get'num$

Inputs a number from the terminal under RADC MULTICS GCOS Encapsulator.

Not used in DDDG - exists for debugging purposes.

proc scan'call(read'call)$

Maintains token'stack; controls scanner invocation. Read'call identifies the type of token wanted.

If read'call is 1, a read token is required. A call is made to get'token, which uses the token stack mechanism to return a read token. Scan'call then checks to see if the token is the start of a comment. If so, it identifies the type of the comment by querying crlf and last'token. Scanner exception 7 is set by the scanner itself upon reading a quote and noting that define'flag is 0. Scan'call pulls in <characters> tokens from scan, one by one, until a quote token is detected. Scan turns exception off, and control loops back to the beginning of scan'call to get a "real" token. After every token, buffer'in is called to put it out to f2'buffer. Note that if the asis toggle is set, no text emanates from scan'call's comment outputting statements.

If the token was not the start of a comment, define'flag is checked. If it is set, <characters> tokens are read into the mdt as part of a new macro definition, which has already been initialized by synthesize. The text is also output to f2'buffer by buffer'in.

If the token returned did not indicate a comment start (i.e., it was not a quote), a check is made to see if it is a macro name. Here, the expand flag is used to decide whether to write the name out to f2'buffer. It is always necessary to preserve the leading blanks before the macro name if the asis toggle is set and the expand toggle is off. If a macro name was seen, scan'call loops back for a "real" token; if not, it returns to its caller.

If the read'call flag is 0, scan'call looks in token'stack for lookahead tokens. If there are none, it calls scan for one, then stacks it. If the token it finds, by either method, is a macro name or part of a comment, the token is negated and re-stacked, and another is fetched. This way, subsequent lookaheads can identify these "non-tokens" and ignore them; but at the same time they are preserved for eventual processing by the read portion of scan'call. Comment of macro processing cannot take place in a lookahead condition - the action taken by the processing routines stands a good chance of being temporally incorrect.

proc toggle'proc(c'token)$

Finds comment toggles inside comment text. C'token is the passed comment text.

Toggle'proc finds occurrences of toggles within comments by searching for "[". If found, it extracts all text between "[" and "]", and then compares this text to entries in array togc. If a match is found, the toggle is turned either on or off, as described in the User's Manual. Toggle values are held in parallel array tog.

proc buf(flag'last,flag'trail)$

Formats and puts out current token text to f2'buffer. Flag'last and flag'trail indicate whether to place blanks before or after the current token text.

If flag'last is 1, buf deletes the last space from the last used line of f2'buffer. If flag'trail is set, buf appends a space to the current token text, bcd. Buf keeps a count of the total number of lines and characters in f2'buffer, which information will be used later by file1'out and file2'out. Most of buf is concerned with correct filling of f2'buffer (no text chopped off, no extra spaces). Its only anomalous behavior occurs when the asis toggle is set. Then, buf concatenates spaces(blanks) to bcd before writing it out. When crlf is set in this mode, buf calls file2'out and file1'out to put out an asis line. It then outputs the concatenated bcd to an empty f2'buffer.

proc buffer'in$

Formats source text and calls buf.

Buffer'in uses formatting information in arrays nopair1, nopair2, nospaa, and nospab plus last'token and token to decide whether to add or subtract spaces from in front or after the current token text. It calls buf with its decision. If the asis toggle is set, buf is called with (0,0).

proc get'token$

Performs read token function for scan'call.

Get'token searches the token stack for a readable token. If one exists, it is read, and the stack updated to delete it. If one does not exist, a call to scan is made. Control then returns to the caller.

proc stack'token(negate)$

Performs lookahead stacking function for scan'call. Negate indicates whether the token should be negated before stacking.

Stack'token simply stacks the current token, negating it if negate is 1.

proc numj(aa)$

Prints out an integer quantity on RADC MULTICS GCOS Encapsulator.

Not used, exists for debugging purposes.

proc compilation'loop$

Performs LALR(k) parsing algorithm.

Uses the tables described in reference 7 to parse the input source code. Compilation'loop varies from the system described in reference 7 in some ways - for instance, a binary search is done on the read states, and the recovery algorithm works differently. Further information about the parsing algorithm is available in any good parsing text. It is recommended that this routine be left strictly alone by system maintenance personnel.

proc print'summary$

Used to print summary of the compilation.

Not used in the current DDDG.

proc main'procedure$

Starts and controls DDDG.

Main'procedure calls compilation'loop, then print'summary. It also outputs the remains of files 1 and 2, which may not have been output.

proc chartype (symbol) $

Chartype is a utility routine which interrogates the type array with the ASCII value of the single character contained in the converted character string symbol. The reasons for making chartype a function involved improved code readability and JOCIT compiler difficulties.

proc getcrd$

Reads in source cards for scan; expands macros from mdt.

If getcrd detects that phony=1, it knows that it must expand the current macro. The index of the current macro (that is, the index into mname, mstart, and mlength) is in macro'name. Getcrd stacks the remains of the current card into ms, which is then pushed down. The macro definition is then read in backwards into the ms, card by card. Until the ms is exhausted, getcrd continues to return cards from it on successive calls.

Other getcrd funcions are setting the crlf flag and correctly orienting the 80-byte input cards within the converted character string called text.

Note that a recursive macro definition will cause the ms to overflow, with an appropriate message.

proc index (string1,string2) $

Returns index of first occurrence of string2 in string1.

Most of index's code is concerned with calculating correct offsets for converted strings.

Index returns 0 if string2 is not in string1.

proc initialize $

Initializes scan databases, critical DDDG variables.

Initialize assigns values to mnemonic variables such a mantissa, star, etc. such that they point at their appropriate match in array VOCAB.

It also initializes the lengths array (used in scan to speed token matching), the type array (same purpose), opens various I/O files, and initializes certain critical quantities.

proc not'letter'or'digit (symb) $

Uses type array to determine whether the first character of symb is a letter or a digit. (see chartype description).

Returns true if symb is not a letter or a digit.

proc scan $

Performs scanning function of LALR parsing algorithm.

Scan separates input source into meaningful language entities called tokens. Tokens are reserved words, user-defined symbols, and special symbols (+,-,*, etc.).

The scanner begins operation by initializing some token data items (blanks, crlf, macro'flag, bcd). It then updates the current text card to ignore the last token read. If there is no more text on the card, a new card is read using getcrd. A case stmt is then entered, on the variable "exception". Usually, exception is zero. In this case, another case stmt is performed on the first character of the new text (using chartype). An examination of the scan source listing or the scan Structured Design Diagram shows how this first comparison actually traps most token possibilities. Case 6 of this internal case stmt picks up blanks, and bumps up the variable "blanks" for each blank it receives. Case 7, which detects "''", sets exception=7 to pick up comment text if define'flag is 0. Case 8 picks up numbers, and case 9 deals with user-defined symbols (identifiers). It also detects macro calls and sets up the necessary mechanics so that getcrd is flagged correctly and called to expand the macro.

If exception=1, the scanner reverts to a mode where it sends <characters> tokens which correspond to text between sets of quotes. This case is used by scan'call to get text from macro definitions. Exception case 2 is used to read in Hollerith constants (using the number'value of the number in the constant). Case 3 sends <characters> tokens between the keywords "direct" and "jovial". It also sets the asis toggle, since this assembly code cannot be flowcharted reasonably. Case 4 picks up text between "v(" and ")" in status constants, and cases 5, 6, 8, 9, and 10 are used to overcome certain JOVIAL constructs which give the scanner mechanism trouble. Case 7 is used for comment text recognition.

4.3.3 Local Declarations

```
proc buf(flag'last,flag'trail)$

 item flag'last b$ ''if set, delete last space in
f2'buffer''

 item flag'trail b$ ''add a space to this token''


proc buffer'in$

 item flag'last b$ ''set this to delete last space''

 item flag'trail b$ ''set this to add space to current
token''

 item tempi1 integer$ ''temporary''


proc chartype (symbol) $

 item chartype integer $ ''result quantity''

 item symbol character $ ''used as index into type array''


proc compilation'loop$

 item guess i 36 s$ ''contains latest binary search guess''

 item ii i 36 s$ ''temporary''
 item jj i 36 s$  ''temporary''
 item overflow h 150 p 14h(stack overflow)$ ''error''
 item state i 36 s$ ''current state number''
 item temp i 36 s$ ''temporary''
 item top i 36 s$ ''upper limit on guess loop''

proc err(aa,bb)$
 item aa h 150$ ''message string''
 item bb i 36 s$ ''unused''

proc file1'out(stmt'type)$
 item stmt'type integer$ ''type of stmt, put in file2''

proc file2'out$

proc fout(nerd) $
 item nerd integer$ ''also type of stmt, put in file2''

proc getcrd$
 item done b$ ''binary flag in while loop''
 item getcrd integer$ ''result string''
 item temph1 character$ ''temporary''
 item tempi2 integer$ ''temporary''
 item tempi1 integer$ ''temporary''

proc get'num$
 item get'num i 36 s$ ''result string''
 item kk i 36 s$ ''temporary''

 item strng h 6$ ''contains hollerith of number''


proc get'token$


proc index (string1,string2) $

 item index integer $ ''result string''

 item length1 integer $ ''length of string 1''

 item length2 integer $ ''length of string 2''

 item  string1  character  $  ''might contain an instance of
string2''

 item string2 character $ ''might be contained in  string1''


proc initialize $

 item  cur'lngth  integer  $  ''length  of  group of current
nonterms''

 item cur'term character $ ''current terminal''

 item digits h 10 p 10h(0123456789) $  ''numbers''

 item index integer $ ''vocab index of current terminal''

 item  letters  h  26  p  26h(abcdefghijklmnopqrstuvwxyz)  $
''letters''

 item new'lngth integer $ ''length of current terminal''

 item special'char h 8 p 8h(=($/.* ') $ ''stand-alones''

 item temp'term h 30 $ ''temporary''


proc main'procedure$


proc noconflict(current'state) $

 item current'state i 36 s$ ''current state''
 item ii i 36 s$ ''temporary''
 item noconflict b$ ''true when state in stack can read
token''

proc not'letter'or'digit (symb) $
 item not'letter'or'digit b $ ''true if symb is "weird"''
 item null b$ ''temporary''
 item symb character $ ''passed character''

proc numj(aa)$
 item aa i 36 s$ ''number to be output''
 item bb h 6$ ''hollerith of number''

proc print'production(prodn,left'stackn,right'stackn)$
 item aa h 150$ ''temporary''
 item bb h 150$ ''temporary''
 item cc h 150$ ''temporary''
 item cc1 h 6$ ''temporary''
 item jj i 36 s$ ''temporary''
 item kk i 36 s$ ''temporary''
 item lenth i 36 s$ ''temporary''
 item left'stackn i 36 s$ ''index into stack of left end of
prod''
 item line h 150$ ''output line built''
 item prodn i 36 s$ ''production number to be dumped''
 item right'stackn i 36 s$ ''see left'stackn''

proc print'summary$
proc recover$
 item end'of'file h 150 p 16h(abort on bad eof)$ ''error''
 item recover i 36 s$ ''contains recovery state''
 item  recover'modify h 150 p 14h(modified parse)$ ''error''
 item tsp i 36 s$ ''new stack pointer in intermal loop''
 item temp i 36 s$ ''temporary''

proc scan $
 item bcd'lng integer $ ''length of current token''
 item done b $ ''while loop flag''
 item key'index integer $ ''index into terminals''
 item temp'char character $ ''temporary''
 item temp'string character $ ''temporary''
 item terminate integer $  ''temporary index into text''

proc scan'call(read'call)$
 item aa character$ ''temporary''
 item bb character$ ''temporary''
 item done b$ ''while loop flag''
 item read'call integer$ ''if 1,read call; if 0,look''
 item temph1 character$ ''temporary''
 item temph2 character$ ''temporary''
 item tempi1 integer$ ''temporary''
 item tempi2 integer$ ''temporary''

proc stack'dump$
 item aa h 150$ ''temporary''
 item line h 150$ ''output line built''
```

```
proc stack'token(negate)$
 item negate integer$ ''if on, stack -token''
 item temph1 character$ ''temporary''

proc synthesize(production'number)$
 item production'number integer$ ''current production''

proc toggle'proc(c'token)$
 item c'token character$ ''comment text string''
 item offon integer$ ''flag determines toggle on and off''
 item temph1 character$ ''temporary''
 item temph2 character$ ''temporary''
 item tempi1 integer$ ''temporary''
 item tempi2 integer$ ''temporary''
```

4.4 Data Files Passed to Phase 2

Data files created by Phase 1 for use by Phase 2 and the Invocation Diagrammer are referred to in all Design Diagrammer documentation as FILE 0, FILE 1, and FILE 2. Their exact JOVIAL definitions can be found in Section 4.3.1.3.

FILE 0 is used by the Invocation Diagrammer to create invocation diagrams. It consists of fixed-length blocks of character strings. The blocks can be deciphered using the following heuristic: if the first entry in a block is "***", then the next name encountered in the block is the name of the procedure which calls all other procedures listed in the block. If the first entry is not "***", then treat this block as though it were a continuation of the last block.

Files 1 and 2 are related, in that FILE 1 points at locations within FILE 2 and contains information about them. Although both files are blocked by the DDDG, they can be thought of as sequential and unblocked. Each FILE 1 record consists of four integer fields: an index into FILE 2, a statement type number, a record counter, and a character count. The index into FILE 2 points at the first FILE 2 record which the FILE 1 record describes. It contains the absolute number of the FILE 2 record, starting from 0. Statement type contains a description of the statement type of the FILE 2 record. A full list of statement types, or "statement tokens", appears in Appendix B. The record counter indicates how many FILE 2 records are involved in the statement token. The character count contains the total number of characters in the statement token.

FILE 2 contains the text of statement tokens found in the input program. Each statement token occupies one FILE 2 record, unless its total length requires more records. The text in FILE 2 is formatted so that left-out or extra spaces in the input text do not appear in the final flowchart. If FILE 2 were treated as a continuous stream of characters, it would be functionally identical to the input program text.

4.5. Design Diagram Generator (DDG) Program Structure

The DDG is responsible for creating design diagrams from the data base created by the DDDG.

The execution of the DDG is broken into two distinct parts.

Part 1 of the DDG accepts user options which describe the desired format of the diagram (see the JSDD Users Manual for a complete description of user options). It builds an intermediate data base (see section 4.5.1) which defines the diagram according to the user supplied specifications.

Part 1 is described in section 4.5.2.

Part 2 of the DDG uses the information in the intermediate data base to extract and format text from File 2 (the program text file created by the DDDG) and produce the design diagram.

Part 2 is described in section 4.5.3.

Section 4.5.4 describes Phase 2 input/output.

Throughout this section, references are made to DDG procedures which are described in Section 4.6.

4.5.1. The Intermediate Data Base

The intermediate data base consists of two disk files: FILE 3 and FILE 4, and a core resident table: GROUP.

FILE 4 contains a set of records for each statement unit contained in FILE 2. The number of records in a set is equal to the number of lines that the statement unit will occupy in the design diagram. A set of FILE 4 records will contain a pointer into FILE 2 and information relating to the length and line breaks of a statement unit.

Section 4.5.1.1 contains a complete description of FILE 4.

FILE 3 is a collection of binary trees which completely describes the design diagram being created. Each FILE 3 record corresponds to a code block (or box) in the design diagram. It contains pointers into FILE 4 indicating what FILE 4 records pertain to the statement units which are to be elements of the code block. Each FILE 3 record also contains block size, type and placement information as well as pointers to other FILE 3 records (which indicate its position within its tree). There is a tree in FILE 3 for each program, procedure declaration, close declaration and stump (see Section 4.5.2.2.3) encountered in the input file.

FILE 3 is described in section 4.5.1.2.

The GROUP table is a linked list which contains the record numbers of the roots of FILE 3's trees. It is described in Section 4.5.1.3.

4.5.1.1. FILE 4 Description

FILE 4 is made up of two types of records: header records and line break records. Each statement unit has one (and only one) header record associated with it.

A header record has three fields :

F2'PTR
: A pointer into FILE 2 describing the location of the statement unit.

LINES'OUT
: The number of lines that the statement unit will occupy in the design diagram.

MAX'LINE'LNGTH
: The length of the longest line of the statement unit.

Line break records describe where a line of text composing a statement unit will be broken in order to comply with the user supplied ST'MAX option (ST'MAX is the maximum number of text characters that can appear on one line of a code block - see JSDD User's Manual). A statement unit will have LINES'OUT-1 line break records associated with it.

Line break records have two fields:

F2'REC
A pointer into FILE 2 indicating the record in which a line break is to be made.

F2'BYTE
The byte in F2'REC at which the break is to be made.

4.5.1.2. FILE 3 Description

Each FILE 3 record contains all information necessary for the placement of a code block in the design diagram.

A FILE 3 record consists of 14 fields:

F4'BEGIN
A pointer to the FILE 4 header record which points at the first statement unit of the code block.

F4'END
A pointer to the last FILE 4 record (header or line break) pertaining to the last statement unit to be displayed in the code block.

STMT'UNIT
The type of code block described by the FILE 3 record. See Appendix B for a list of code block (statement) types.

START'COL
The page column on which the code block's display will begin.

BLOCK'WIDTH
The length of the longest line of text to be displayed in the code block.

STOP'COL
The page column on which the code block's display will stop.

START'LINE
The line on which the code block's display will begin.

LINES
The number of lines spanned by the code block (not including lines for page headings which may be embedded in the code block).

STOP'LINE
The line on which the code block's display will end.

H'PTR
Horizontal pointer. If H'PTR is non-zero, the FILE 3 record containing it is a control phrase (see Appendix B). H'PTR points to the FILE 3 record which begins the scope of control phrase. The code block defined by the FILE 3 record to which H'PTR points will appear to the right of the code block whose FILE 3 record contains the H'PTR. A zero H'PTR

indicates that there is no horizontal scope for the record containing it.

V'PTR
: Vertical pointer. A pointer to the FILE 3 record which describes the code block that will appear under the code block whose FILE 3 record contains the V'PTR.

BACK'H
: The inverse of H'PTR.

BACK'V
: The inverse of V'PTR. Each record in a FILE 3 tree can have at most one ''parent'', so BACK'H and BACK'V cannot both be non-zero.

MIDPT
: The line of the code block's display which is its midpoint. This field is meaningful only for control phrases.

4.5.1.3. GROUP Table Description

GROUP contains a set of entries for each of FILE 3's trees. Its entries are:

F3'REC
: A pointer to the root of a FILE 3 tree.

FROM'PAGE
: If the tree indexed by F3'REC describes a stump, then this entry will contain the number of the page of the diagram on which the stump was referenced.

NEXT
: The link to the next tree's pointer in the diagram sequence.

PAGE'REF
: In Part 1, this entry will contain the number of pages that the tree will occupy. Part 2 will compute from this the page number on which the display of the tree will begin.

PROC'NAME
: The name of the program, procedure or close which the tree describes. If the tree describes a stump, then PROC'NAME is filled with blanks.

The order in which GROUP's entries are linked together will determine the order in which the trees to which they point will appear on the diagram. That ordering is in accordance with the following rules:
1) A stump subtree (see Section 4.5.2.2.3) will appear immediately after the tree to which it is logically connected.
2) The tree describing the main program will appear first.
3) Procedure declarations and close declarations will appear in the order in which their declarations appear in the input program.

4.5.2. Part I of the DDG

Part I of the DDG generates the disk resident intermediate data base and also generates a core resident intermediate data base (a table called GROUP). Although generation of the three units of the intermediate data base takes place in parallel, it is convenient to discuss these functions separately. Section 4.5.2.1 describes FILE 4 generation and Section 4.5.2.2 discusses generation of FILE3 and GROUP.

4.5.2.1. FILE 4 Generation

FILE 4 contains the information needed to break up statement units into text lines whose lengths are compatible with the user supplied ST'MAX option. In order to accomplish this, FILE 1 must be read and its FILE 2 pointers must be followed to perform text analysis.

Upon reading a FILE 1 record, a mapping is made of the FILE 1 STMT'TOKEN field onto one of eleven values of STMT'TYPE (this mapping is performed by the function BOX'MAP).

If STMT'TYPE contains a value equal to CONTROL'1, CONTROL'2, CONTROL'3, or CONTROL'4 then the statement unit to which it refers is a control phrase (e.g., IF CONDITION $ is a control phrase whose STMT'TYPE is CONTROL'2). Each control phrase has associated with it a block of code which is referred to as its scope. The scope of a control phrase is that block of code whose execution is controlled by the control phrase. Following the control phrase and its scope, a dummy statement unit having the STMT'TYPE END'SCOPE appears. END'SCOPE signals the termination of a control phrase's scope. END'SCOPE is referred to as a dummy statement unit because it is not a part of the grammar which defines the syntax of JOVIAL J3.

If STMT'TYPE indicates that the statement unit to which the current FILE 1 record refers is a printing statement unit, then a set of FILE 4 records must be created for the statement unit. Otherwise, no FILE 4 records are created.

A set of FILE 4 records consists of one header record followed by any number (including zero) line break records (see Section 4.5.1.1).

If STMT'LNGTH (from FILE 1) does not exceed ST'MAX, then the header record is the only record in the set. The statement unit to which it refers will appear on a single line of the diagram. LINES'OUT is set to 1 and MAX'LINE'LNGTH is set to the length of the statement unit.

If STMT'LNGTH exceeds ST'MAX, then the statement unit will appear on two or more lines of the diagram. An attempt is made to break the statement unit at a space. However, if this is not possible, the statement unit is broken such that the length of the line of statement unit text is equal to ST'MAX.

The statement unit is actually broken by setting the values of the pointers (F2'REC and F2'BYTE) in the line break record to the record number and byte (in FILE 2) at which the break will occur.

Each line break record corresponds to one break of the statement unit's text.

LINES'OUT and MAX'LINE'LNGTH are updated after each line break record is created.

The procedures controlling FILE 4 creation are BOX'MAP and CREATE'FILE4'RECS.

4.5.2.2. FILE 3 and GROUP Table Generation

FILE 3 contains binary trees which define the printing sequence of the code blocks. Each FILE 3 record contains all the information necessary to print the code blocks on the diagram. The table GROUP contains pointers to the root of each of the trees in FILE 3.

FILE 3 generation can best be understood as a three part operation where the three parts are record initialization, record continuation and record closure. These operations are discussed in Sections 4.5.2.2.1, 4.5.2.2.2 and 4.5.2.2.3, respectively.

4.5.2.2.1. Record Initialization

A new FILE 3 record is initialized when it becomes evident that a new code block is needed (see Section 4.5.2.2.3 for the conditions).

Record initialization consists of entering available information into the new record and zeroing fields that require information which is not yet available.

Immediately upon recognizing the need for creating a new record, the following fields may be assigned: F4'BEGIN, F4'END, STMT'UNIT, BLOCK'WIDTH, LINES, BACK'H and BACK'V. Of these, F4'END, BLOCK'WIDTH and LINES are subject to revision if the record is continued.

Record initialization is handled by the procedures UPDATE'FILE3, CREATE'H'PTR'REC, CREATE'V'PTR'REC and INITIATE'RECORD.

4.5.2.2.2. Record Continuation

A record is said to be continued if the next set of FILE 4 records refers to a statement unit which should be included in the current code block.

Record continuation will occur under the following two conditions:
1) The next set of FILE 4 records refers to a 'type-1' comment. (See Appendix B)
2) The FILE3 record does not describe a control phrase and the next set of FILE 4 records refers to a statement unit of the same type as those included in the FILE 3 record.

Record continuation consists of resetting F4'END so that the next set of FILE 4 records is included in the FILE 3 record. The BLOCK'WIDTH and LINES fields of the FILE3 record are also updated accordingly.

Record continuation is controlled by the procedures UPDATE'FILE3 and CONTINUE'BOX.

4.5.2.2.3. Record Closure

Record closure is the operation which assigns those fields in the FILE 3 record which define the positioning of the code block on the diagram (they are START'COL, STOP'COL, START'LINE, STOP'LINE and MIDPT).

Also, record closure may cause a new FILE 3 record to be initialized (as described in Section 4.5.2.2.1).

A record is closed when the decision is made that no more statement units will appear in the code block defined by the record. This decision is based on the STMT'TYPE of the statement unit to which the next set of FILE 4 records refers.

A FILE 3 record will be closed upon satisfying either of the following conditions:
1) The dummy statement unit END'SCOPE is encountered.
2) The next set of FILE 4 records doesn't satisfy either condition for record continuation (see Section 4.5.2.2.2).

Satisfaction of closure condition 1 indicates that the scope of a control phrase is being terminated. The record being closed is the last record within the scope of the most recently created control phrase record. The functions performed at this time are the positioning of the code block on the diagram and a return to the record defining the control phrase whose scope is being terminated.

Returning to the control phrase is accomplished by following BACK'H and BACK'V pointers until a record that defines a control phrase is found. There are two possibilities at this point. Either an END'SCOPE will be encountered (in which case more backup will be performed) or a new record will be initialized and pointed to by the control phrase record's V'PTR.

Satisfaction of closure condition 2 indicates that a new code block (and a new FILE 3 record) must be initialized. It must be pointed to by either the H'PTR or V'PTR of the record being closed.

The determination of whether H'PTR or V'PTR will point to the new record is made by examining the value of the STMT'UNIT field of the record being closed. If STMT'UNIT is equal to CONTROL'2, CONTROL'3 or CONTROL'4, then the record being closed is a control phrase which must point at a horizontal scope. If H'PTR is zero, it is set to point at the new record. The BACK'H field of the new record is set to point at the record being closed.

In all other cases, V'PTR will point to the new record (and BACK'V of the new record will point back).

Regardless of which closure condition was satisfied, the code block defined by the record undergoing closure must be placed in the diagram. This function is performed by the procedure PLACE.

PLACE finds the father of the code block by following either the BACK'H or BACK'V pointer (if one of them is non-zero). If the father is pointed to by the BACK'H pointer, then PLACE will attempt to position the code block to the right of the code block defined by the father. If the father is pointed to by BACK'V, then PLACE will attempt to position the code block under the code block defined by the father.

If neither BACK'H nor BACK'V is non-zero, then the record being placed must have a STMT'UNIT field equal to CONTROL'1. This means that the record describes a <PROGRAM HEAD>, <PROC DESCRIPTOR> or <CLOSE HEAD> and that it is the root of one of the FILE 3 trees. There will be no way to access this tree root from any other tree in the file, so the record number of the root must be inserted into the GROUP table. This function is performed by INSERT.

Code block placement fails if an attempt is made to assign to STOP'COL a value which exceeds PAGE'WIDTH. The code block is referred to as a stump. A stump is defined to be a subtree which does not fit in the available number of page columns.

After a stump is detected the procedure RESOLVE'STUMP finds the root of the stump. The root of a stump is not necessarily the code block that caused the placement failure. Type-2 comments and CONTROL'3 code blocks are not legal stump roots. If the code block which caused the placement failure is not a legal stump root, then BACK'H and BACK'V pointers are followed until a legal stump root is found. Care is taken to assure that the position of the father of the stump root on the diagram will allow the display of a stump reference box.

When the stump root is found, the H'PTR or V'PTR by which its father accesses it is negated in order to indicate that the code block is a stump root. The BACK'H or BACK'V pointer by which the stump root accesses its father is also negated.

The stump root and all of its descendents must be positioned on the diagram. This is done by repeated invocations of PLACE from within a tree traversal algorithm (see Appendix A).

Stump roots are inserted into the GROUP table in a manner similar to tree roots. Even though each stump root is accessible from some record in one of the FILE 3 trees, the stump root is treated as the root of a separate tree.

It should also be noted here that all values placed in the START'LINE, STOP'LINE and MIDPT fields of FILE 3 records are relative to the trees to which they belong. That is, every record being pointed to by a GROUP table entry is treated as if the tree of which it is the root is the only tree described in FILE 3. Relative line numbering is necessary because initiating a new tree does not guarantee that the old tree has been completed. The number of lines that will be spanned by code blocks described by the old tree will not be known until its processing has been completed. Relative line numbers will be made absolute in Part 2.

When an insertion is made into the GROUP table, the line information relating to the current tree is pushed onto LAYOUT'STACK so that it may be retrieved when the new tree's processing has been completed.

4.5.3 Part 2 of the DDG

Part 2 of the DDG is the output processor. It traverses the FILE 3 trees generated by Part 1 and outputs a code block for each FILE 3 record.

Each FILE 3 record having a STMT'UNIT field equal to CONTROL'1 (i.e. each FILE 3 record which describes a <PROGRAM HEAD>, <PROC DESCRIPTOR> or <CLOSE HEAD>) is the root of a FILE 3 tree. Also, any record having a negative BACK'H or BACK'V pointer is the root of a FILE 3 tree (i.e. such records are stump roots. See Section 4.5.2.2.3). Each tree in FILE 3 has an entry in GROUP whose F3'REC item points to its root. The entries in GROUP are linked together so that they may threaded through in the order in which the trees will appear on the design diagram.

The first major task performed by Part 2 of the DDG is the assignment of a reference number to the PAGE'REF item in each entry of GROUP.

If the HEADING option is on, then the reference numbers will be page numbers. The page numbers are calculated by following the links through GROUP and accumulating the values that Part 1 stored in each entry's PAGE'REF item. The values stored there by Part 1 indicate the number of pages on the diagram that will be occupied by the tree to which the GROUP entry refers. The accumulated page totals are then entered into each GROUP entry's PAGE'REF item.

If the TABLE'OF'CONTENTS option is also on, then a table of contents entry is generated (by the procedure GENERATE'CONTENTS'ENTRY) for each module referred to by a GROUP entry. A module is defined to be a program, procedure declaration or close declaration. A module can be distinguished from a stump in the GROUP table because it has a non-blank PROC'NAME. (Stumps do not have entries in the table of contents).

If the HEADING option is off, then there will be no page numbers in the diagram. However, stumps will be numbered according to the order of their appearance in the diagram. The stump numbers will be stored in the GROUP entries' PAGE'REF items.

Assignment of reference numbers is performed by the Part 2 procedure COMPUTE'PAGE'NUMBERS.

After reference numbers are assigned to each entry in GROUP, the outputting of the design diagram begins.

Each of the FILE 3 trees is processed independently and in the order determined by Part I (see Section 4.5.1.3). The next tree to be processed is accessed by the NEXT pointer of the current tree's GROUP entry (NEXT(O) accesses the first tree). When an attempt is made to follow a zero link, then the tree processing operation is complete.

The processing of a FILE 3 tree begins with the invocation of the procedure ADVANCE'PAGE which sets LINE'NO (the number of the current output line) to point at the top of the next unused page. The new value of LINE'NO becomes the displacement to which all of the relative line numbers in the FILE 3 tree are added in order to obtain absolute line numbers.

The tree is traversed by the method described in Appendix A. Each FILE 3 record describes a code block having one of the four box types described in Section 4.5.3.1. Five basic operations are performed on each record of the tree. These operations are: connecting the code block, outputtting the code block top, extracting the input program text, constructing the code block lines and outputting the code block bottom.

Connecting the code block is the operation which draws a connecting line between the current code block and its father. This operation is described more completely in Section 4.5.3.2.

Outputting the code block top is the operation of outputting the top of the code block's box on the output line designated by the START'LINE field of the FILE 3 record. This operation is performed by the procedure OUTPUT'BOX'TOP.

Extracting the input program text is performed by the function EXTRACT'TEXT. The purpose of the function is to extract the contents of FILE 2 according to the specifications contained in FILE 4. This function is discussed more fully in Section 4.5.3.3.

Construction of code block lines is the operation which embeds a line of extracted input program text in the code block's box and outputs the resulting string on the appropriate output line. CONSTRUCT'LINE, the procedure controlling this operation, is also responsible for handling double spacing (if the DOUBLE'SPACE option is on) and for invoking the heading outputter (OUTPUT'HEADING) if the top of a page is encountered (and if the HEADING option is in effect).

Outputting the code block bottom is almost identical to outputting the code block top. It is performed by the procedure OUTPUT'BOX'BOTTOM. However, OUTPUT'BOX'BOTTOM also performs some buffer optimization which is described in Appendix C.

If a record having a negative H'PTR or V'PTR is found while traversing a tree, then that record is the father of a stump root (see Section 4.5.2.2.3). All such records must have stump reference displays indicating the number of the page on which the stump root's subtree will appear (if HEADING is on) or the stump sequence number (otherwise).

Stump reference displays are constructed and output by the procedure DISPLAY'STUMP'REF. Basically, this procedure draws a box to the right or under the stump root's father (depending upon whether H'PTR or V'PTR is negative). The contents of the PAGE'REF item of the GROUP entry (which points at the stump) is then displayed in the box. If the HEADING option is in effect, then the number of the page on which the stump is referenced is stored in the FROM'PAGE item of the GROUP entry.

4.5.3.1 Code Block Formats

There are four code block formats in the JSDD diagrams.

Code blocks whose FILE 3 records have a STMT'UNIT field equal to SEQ or PGM'TAIL appear as rectangles:

```
**********
*        *
**********
```

Code blocks defined by COMMENT'2 FILE 3 records appear as follows when accessed by a V'PTR:

```
/
/ ''COMMENT TEXT''
/
```

When accessed by an H'PTR, COMMENT'2 code blocks appear:

```
-+
 / ''COMMENT TEXT''
 /
```

CONTROL'1, CONTROL'2 and CONTROL'4 code blocks appear in the following manner:

```
**********
*        *
**********
```

CONTROL'3 code blocks have the following appearance:

```
          **********
       +-*         *
          **********
```

4.5.3.2 Connecting a Code BLock

The procedure CONNECT'BOXES is responsible for drawing connecting lines between a code block and its father (if the code block has a father).

If the current code block has no father then no action is taken.

If the current code block has a negative BACK'H or BACK'V pointer, then it is a stump root and it must appear under a stump reference display box. Such a display is constructed by drawing a rectangular box and placing the contents of the current GROUP entry's PAGE'REF in the box. If the HEADING option is in effect, then the contents of FROM'PAGE are also displayed in the box.

In all other cases, the current code block has a father to which it must be connected. CONNECT'BOXES calculates the X and Y coordinates (in terms of column and line) of the end points of the connecting lines and passes them to the procedure DRAW'LINE which draws them.

4.5.3.3 Extracting the Input Program Text

Each statement unit appearing in FILE 2 has a set of FILE 4 records associated with it. This set of records points to the statement unit in FILE 2 and defines the way in which it will appear in the diagram (see Sections 4.5.1.1 and 4.5.2.1). The function EXTRACT'TEXT operates on FILE 2 with the information in FILE 4 .

If the Header record of the set contains a LINES'OUT field having a value of one, then the statement unit to which it points will appear on one line of the diagram and the entire FILE 2 record is returned.

If LINES'OUT is greater than one, then LINES'OUT invocations of EXTRACT'TEXT must be made. Each invocation extracts and returns the text between the bytes indicated by consecutive records of the set of FILE 4 records.

4.5.4 Phase 2 Input/Output

All of the files generated by Phase 2 (FILE 3, FILE 4 and the output file, PUTOUT) are treated as direct access files. This is necessary because of the frequent updating and back-up required by the JSDD. Since no direct access facilities exist for JOVIAL output files, Phase 2 uses a double-buffering system to manage its files.

The double-buffering requires that Phase 2 maintain two copies of each of its files. One of these copies is flagged as being the most recent version. A block (or in the case of PUTOUT, a collection) of records is kept as an in-core buffer.

All Phase 2 files are accessible only by the ACCESS functions (ACCESS3, ACCESS4 and ACCESS'OUT). The ACCESS functions take one parameter -- the absolute record number of the record to be accessed. Among other things, the ACCESS functions read the block containing the desired record into the buffer (if it exists and is not already in core) and return the index into the buffer of the desired record.

A write operation has the immediate effect of changing the contents of the in-core buffer (not of the block on disk). Any changes made to the contents of the buffers cause a write switch associated with the file pair to be activated. If a call to an ACCESS function requests a record not currently in core, then before the block containing the desired record can be brought into core, the write switch must be evaluated. If it is on, then the current buffer must be written out to the file. However, in order to accomplish this, all blocks in the most recent version (and the contents of the buffer) must be transferred over to the older version of the file. The old version is then flagged as being the most recent version.

The transfer operations are performed by the procedures TRANSFER'WRITE'3, TRANSFER'WRITE'4 and TRANSFER'WRITE'OUT.

Appendix C describes current and possible future optimizations of the double buffering system.

4.6 Phase 2 Modules, Variables and Constants

This section contains a list of DDG variables and procedures. Each item in the list is accompanied by a brief description of its function and/or meaning. Following the list of procedures are the three compools OPT and DEBUG which were used to produce the design diagram of the DDG. (SPOOL is also used by the DDG. See Section 4.1.5).

The DDG global declarations are listed in Section 4.6.1. Procedures (and their local variables) are listed in Section 4.6.2. OPT and DEBUG are listed in Section 4.6.3.

In Sections 4.6.1 and 4.6.2 the following DEFINE DIRECTIVES are in effect:

```
define    character ''h 150'' $
define    f1'blksiz ''78'' $
define    f2'blksiz ''12'' $
define    f3'blksiz ''22'' $
define    f4'blksiz ''105'' $
define    false     ''0'' $
define    integer   ''i 36 s'' $
define    out'buf'size        ''1000'' $
define    true      ''1'' $
```

4.6.1 DDG Global Declarations

This section contains a list of all global variables declared in the DDG. In the descriptions following the declarations, PT1 indicates that the declared variable is used only in Part 1 of the DDG and PT2 indicates that it is used only in Part 2. All variables that are designated as neither PT1 or PT2 can be assumed to be used in both DDG parts.

```
item bottom'line integer $ ''PT2 stores STOP'LINE of CUR'REC
''

item block'delim integer p 5 $ '' a STMT'TYPE constant ''

item box'tail integer p 2$ ''length of CONTROL'3 tail ''

item comment'1 integer p 2 $ '' a STMT'TYPE constant''

item comment'2 integer p 3 $ '' a STMT'TYPE constant''

 ''CONTROL'1 thru CONTROL'4 are STMT'TYPE constants for ''
 ''control phrases.''

item control'1 integer p 8 $ ''PROGRAM, PROC or CLOSE head''

item control'2 integer p 9 $ ''IF, FOR or DO clause''

item control'3 integer p 10 $ ''IFEITH clause or CASE HEAD''

item control'4 integer p 11 $ ''ORIF clause or INSTANCE ''

item  cur'group  integer p 0 $ ''index into group of current
tree''
```

```
item cur'rec integer p 0 $ ''current FILE 3 record ''

item delim'comment integer p 4 $ '' STMT'TYPE constant ''

item disp integer $ ''PT2  displacement  for  relative  line
numbering''

item  display'lines integer p 3 $ ''lines spanned by a stump
ref box''

item display'width integer p  6 $''width of stump ref  box''

item end'scope integer p 1 $''STMT'TYPE constant ''

item eofile b p 0 $''FILE 1 end of file flag''

item  extra'block  b  p  false  $  ''flags extra PUTOUT file
block''

  '' F1'BUF is the FILE 1 buffer  see Section 4.4 ''

table f1'buf r f1'blksiz s n $
 begin
 item field1 integer $
 item field2 integer $
 item field3 integer $
 item field4 integer $
 end

 '' F2'BUF is the FILE 2 buffer see Section 4.4''

table f2'buf r f2'blksiz s n $
 begin
 item f2'line character $
 end

item f3'avail integer p 1 $ ''the next empty FILE  3  record
''

 ''F3'BUF is the FILE 3 buffer see Section 4.5.1 ''

table f3'buf r f3'blksiz s n $
 begin
 item f4'begin integer $
 item f4'end integer $
 item stmt'unit integer $
 item start'col integer $
 item block'width integer $
 item midpt integer $
 item stop'col integer $
 item start'line integer $
 item lines integer $
 item stop'line integer $
```

```
    item h'ptr integer $
    item v'ptr integer $
    item back'h integer $
    item back'v integer $
    end

  ''F4'BUF  is the FILE 4 buffer: see Section 4.5.1 ''

table f4'buf r f4'blksiz s n $
 begin
 item f2'ptr integer $
 item lines'out integer $
 item max'line'lngth integer $
 item f2'rec integer $
 item f2'byte integer $
  '' overlay the disjoint fields''
 overlay lines'out = f2'rec $
 overlay max'line'lngth = f2'byte $
 end

  '' GROUP is the FILE 3 tree pointer. See Section 4.5.1 ''

table group v 500 p n $
 begin
 item f3'rec integer $
 item from'page integer $
 item next integer $
 item page'ref integer $
 item proc'name character $
 end

item  f2'recs  integer  $  ''no. of FILE 2 recs spanned by a
stmt unit''

item f1'blk integer p -1 $ ''number of FILE 1 block in  core
''

item  f2'blk integer p -1 $ ''number of FILE 2 block in core
''

item f3'blk integer p -1 $ ''number  of  FILE  3  block   in
core''

item  f4'blk integer p -1 $ ''number of FILE 4 block in core
''

item f3'empty integer p 0 $ ''next empty FILE 3 block ''

item f4'empty integer p 0 $ ''next empty FILE 4 block ''

item f4'start integer $ ''PT2 stores F4'BEGIN of CUR'REC  ''
item f4'stop integer $ ''PT2 stores F4'END of CUR'REC ''

file  filel  b 10000 r 313 v(a) v(b) v(c) v(d) v(eofl)  11 $
''FILE 1''

file file2 h 10000 v 1806 13 $ ''FILE 2''

file file3'1 b 10000 v 309 14 $ ''Version 1 FILE 3''

file file3'2 b 10000 v 309 18 $ ''Version 2 FILE 3''

file file4'1 b 10000 r 316 15 $ ''Version 1 FILE 4''

file file4'2 b 10000 r 316 19 $ '' Version 2 FILE 4''


item file2'index integer $ ''PT1 index of stmt unit in  FILE
2 ''

 ''FILE3'INCLUSION is a flag indicating whether a FILE 3 rec
should be created for the current FILE 1 rec (PT1)''

item file3'inclusion b $

file final'out h 20000 v 150 17 $ '' output file''

item  first'4  integer  $ ''  PT1  index of 1st FILE 4 rec of
set ''

item  first'invoc  b. p  true  $  ''flags  first  call  to
EXTRACT'TEXT''

item  group'avail  integer p 1 $ ''index of next empty space
in group ''

item group'max integer p 499 $ '' the size of GROUP ''

array group'stack 100 integer $''PT2  for  storage  of  tree
history''

item  headroom  integer  p  4  $  ''lines  spanned  by  page
heading''

item h'father integer $ ''PT2 stores BACK'V of CUR'REC ''

item h'son integer $ ''PT2 stores H'PTR of CUR'REC ''

item h'space'1 integer p 4 $ '' horizontal spacing  constant
''

item  h'space'2 integer p 2 $ '' horizontal spacing constant
''
```

item last'4 integer p -1 $ ''PT1 index of last FILE4 rec of a set''

item last'f1 integer p -1 $ ''PT1 index of last FILE1 rec read''

''LAST'LINE is used in Part 1 to store the last relative line number in the current tree. In Part 2 it is the absolute last line which has been output. ''

item last'line integer $

item last'proc integer p -1 $ ''PT1 index in GROUP of the last proc ''

item last'stump integer p 0 $ ''PT1 index in GROUP of the last stump ''

array layout'stack 100 integer $ '' stores history of LAST'LINE''

item layout'stack'max integer p 99 $ ''capacity of LAYOUT'STACK''

item layout'stack'top integer p 0 $ '' current top of LAYOUT'STACK''

item left'col integer $ ''PT2 START'COL of CUR'REC''

item line'no integer $ ''PT2 the number of the line being output''

item max'foutput integer p -1 $ ''PT2 index of last PUTOUT block on disk''

item message character $ '' stores error and debug messages''

item midpoint integer $ ''stores MIDPT of CUR'REC''

''NAME'BYTE is the byte of header at which the module name prints''

item name'byte integer $

item new'file3 b p 0 $ ''flags FILE3'1 or FILE3'2 as most recent''

item new'file4 b p 0 $ ''PT1 flags FILE4'1 or FILE4'2 as most recent''

item new'out b p 0 $''PT2 flags PUTOUT'1 or PUTOUT'2 as most recent''

```
item  new'text  character $ ''PT2 stores text extracted from
FILE2''

item null'scope b p 0 $''PT1  flags  control  phrase's  null
scope''

item out'blk integer p -1 $ ''PT2 PUTOUT block in core ''

 ''OUT'LINE is PUTOUT's buffer ''

array out'line out'buf'size character $

item  page'byte  integer $''PT2 header byte at which page'no
will print''

item page'no integer $ ''PT2 number of current page ''

item past'mid b $ '' PT2 flag controls printing  of  pointed
boxes''

item pgm'tail integer p 6 $ ''STMT'TYPE constant''

item  proc'flag  b  p  0  $ ''PT1 flag on if creating a proc
tree''

item proc'root integer p 0 $ ''PT1 flags  GROUP  item  as  a
proc head''

array proc'stack 5 integer $ ''PT1 stores proc history''

item  proc'stack'max  integer  p  4  $  ''  PT1  PROC'STACK
capacity''

item proc'stack'top integer p 0 $ '' PT  1  current  top  of

PROC'STACK''

item readsw b p false $ ''a read-only switch for FILE4''

item  proc'count  integer  p 0 $''PT2 the number of procs in
the program ''

file putout'1 h 20000 v 150 16 $ ''Version 1 tmp output''

file putout'2 h 20000 v 150 20 $ ''Version 2 tmp output''

item recs'in'blk1 integer $''number of records in the  FILE1
block''

item  recs'in'blk2  integer  $ ''number of recs in the FILE2
block''
```

```
item recs'in'blk3 integer p 0 $''number  of  recs  in  FILE3
block''

item  recs'in'blk4  integer  p 0 $ ''number of recs in FILE4
block''

item right'col integer $ ''PT2 stores STOP'COL of  CUR'REC''

item  right'pos  integer $''PT2 monitors printing of pointed
boxes''

item seq integer p 7 $ ''STMT'TYPE constant ''

item skip b $ ''PT2 controls double spacing''

item  single'space  b  $  ''  PT2  always  set  to  NOT
DOUBLE'SPACE''

item son'top integer $ ''PT2 START'LINE of CUR'REC's V'SON''

item stmt'token integer $'' the type of a statement unit''

item stmt'type integer $ ''a statement unit type''

item  stmt'lngth integer $ ''PT1 stores the length of a stmt
unit''

item stump'found b p 0 $ ''PT1 flag set if a stump has  been
detected''

item  stump'root  integer  p  1 $ ''PT1 flags GROUP entry as
stump head''

item tempc character $ '' temp used for characters''

item tempi integer $ '' temp for integers''

item top'line integer $ ''PT2  START'LINE of CUR'REC''

item t'mess character ''temp used for debug messages '' $

array  traverse'stack  200  integer  $  ''stack  for  tree
traversal''

item    traverse'top    integer    $''    current  top  of
TRAVERSE'STACK''

item v'father integer $ '' PT2 BACK'V of CUR'REC''

item v'son integer $ ''PT2 V'PTR of CUR'REC''

item v'space integer p 3 $ ''vertical spacing constant''
```

item width integer $ ''PT1 stores width of current text line or box''

item write3 b $ '' FILE 3's write switch''

item write4 b $ ''FILE 4's write switch''

4.6.2 DDG PROCEDURES

The following is the list of procedures declared in the DDG. In addition to these procedures, the string package is also used by the DDG (see Section 4.1).

proc access1 (rec'no) $

''The function ACCESS1 provides the interface for FILE 1. Its parameter is an absolute record number. ACCESS1 reads the appropriate block of FILE 1 records into core and returns the index into FILE 1's buffer of the desired record.''

item access1 integer $ ''index into F1'BUF of record''

item block'no integer $ '' FILE1 block in which record resides''

item rec'no integer $ ''absolute number of the record''

proc access2 (rec'no) $

''ACCESS2 accepts an absolute record number and makes sure that the FILE2 block containing that record is in core. ACCESS2 returns the index into F2'BUF of the record. ''

item access2 integer $ ''the index into F2'BUF of the record''

item block'no integer $ ''the FILE2 block containing the record''

item rec'no integer $''the absolute number of the file2 record''

proc access3 (rec'no) $

'' The function ACCESS3 is the interface for FILE3. Its parameter is the absolute record number of a FILE 3 record. ACCESS3 determines if the block containing the record is in core. If it is, then, the index into the buffer of the record is returned. If the block is exists but is not in core, then WRITE3 is examined. If WRITE3 is on (equal to 1) then, the buffer must be written to disk (by TRANSFER'WRITE3). ACCESS3 reads the desired block into the buffer and returns the index into the buffer of the record. If the block does not exist then ACCESS3 returns the index into the buffer of the record as if the block did exist, and sets RECS'IN'BLK3 to that value.''

```
item access3 integer $ '' the index into F3'BUF of the
record''

item block'no integer $ ''the FILE3 block containing the
record''

item rec'no integer $ ''the absolute number of the
record''


proc access4 (rec'no) $
```

'' The function ACCESS4 is the FILE 4 interface. It is the same as ACCESS3 except that it doesn't call TRANSFER'WRITE4 if READSW is on. ''

```
item access4 integer $ '' the index into F4'BUF of the
record''

item block'no integer $ '' the FILE4 block containing the
record''

item rec'no integer $ ''the absolute number of the record''


proc access'out (rec'no) $
```

'' The function ACCESS'OUT is the interface for the PUTOUT files. Its purpose and operation are analagous to those of access3. ''

```
item access'out integer $'' index into LINES'OUT of the
record''

item block'no integer $''the 'block' containing the
record''
```

item rec′no integer $′′ absolute number of the record in putout′′

proc advance′page $

′′ The procedure ADVANCE′PAGE sets LINE′NO to the number of the top line of the next empty page . LINE′NO then becomes the displacement used for calculating absolute line numbers from relative ones. ′′

item line integer $ ′′ displacement of LINE′NO on current page′′

item page integer $ ′′a dummy variable used in calling REMQUO′′

proc box′map (stmt′type) $

′′The function BOX′MAP accepts STMT′TOKENs used by Phase 1 and maps them onto statement types used by Phase2.′′

item box′map integer $ ′′the phase2 statement type′′

item stmt′type integer $ ′′the input parameter′′

proc byte′em$

′′The procedure BYTE′EM reads the diagram from either PUTOUT1 or PUTOUT2, truncates it and writes it to FINAL′OUT.′′

item line′count integer $′′number of lines in this output group′′

item page integer $ ′′dummy used in calling REMQUO′′

item temph1 character$ ′′ a temporary character variable′′
proc close′rec (term′rec) $

′′ The procedure CLOSE′REC assigns the LINES and BLOCK′WIDTH fields of FILE 3 records. It calls PLACE, which completes the Part 1 processing of the FILE 3 record. ′′

item stmt′type integer $ ′′ the Part 2 stmt type of the record′′

item term′rec integer $′′record number of the record being closed′′

```
 item tempi1 integer $ ''temporary integer ''

 item tempi2 integer $ ''temporary integer ''


proc compute'page'numbers $

 ''The  procedure  COMPUTE'PAGE'NUMBERS  assigns  reference
numbers  to  stumps and if HEADING and TABLE'OF'CONTENTS are
on, calls GENERATE'HEADER'ENTRY to enter a module name  into
the table of contents. ''

 item abs'page'no integer $''number of page on which modules
start''

 item  contents'pages  integer $ ''pages spanned by contents
table''

 item cum'pages integer  $ ''cumulative page count''

 item stump'count integer p 0 $ ''cumulative stump count  ''


proc connect'boxes $

 ''  The procedure CONNECT'BOXES determines how to connect a
code block to its father (if it has one).  If the code block
is a stump then  CONNECT'BOXES  draws  the  stump  reference
display box for the stump. ''

 item display'top integer $ '' top line of the display box''

 item  father  integer  $  ''record  number  of code block's
father''

 item father'bottom integer $ ''father's bottom line''

 item father'mid integer $ ''father's midpoint''

 item father'top integer $ ''father's top line''

 item father'type integer $ ''father's type ''

 item horiz b $ '' type of connection. horiz or vert''

 item line integer $  ''LINE'NO's  displacement  on  current
page''

 item lngth integer $ ''used to store length of strings''

 item page integer $ ''dummy used to call REMQUO ''
```

item x1 integer $ ''x coord of connecting line's start''

item x2 integer $ '' x coord of connecting line's end''

item y1 integer $ '' y coord of connecting line's start ''

item y2 integer $ '' y coord of connecting line's end''

proc construct'line $

'' The procedure CONSTRUCT'LINE embeds the line of input text provided by the procedure EXTRACT'TEXT in the sides of the box in which it will appear. CONSTRUCT'LINE also is responsible for outputting blank text lines if the DOUBLE'SPACE option is in effect. Double spacing is handled by iterating through the routine INC times where INC is set according to the value of SKIP, the blank line flag . ''

item inc integer $ ''the uppper bound for the output loop''

item line integer $ ''LINE'NO's displacement on current page''

item lngth integer $ ''used to store string lengths''

item page integer $ ''dummy variable used for call to REMQUO''

item real'start integer $''stores the column in which text starts''

proc continue'box (cont'rec) $

''The procedure CONTINUE'BOX appends compatible statement units to the current FILE 3 record. A compatible statement unit is one which can appear in the same code block with the statement units which have already been placed there. ''

item cont'rec integer $ '' rec no. of continued block. ''

proc create'file4'recs $

'' The procedure CREATE'FILE4'RECS creates a set of file 4 recs for a statement unit if it is approprate to do so. The primary function of CREATE'FILE4'RECS is to break up statement units (in FILE 2) so that they may be displayed in accordance with the value of ST'MAX. The procedure attempts to break up statement units at the space closest to ST'MAX.

If there is no blank, then the break is made at ST'MAX. A set of FILE 4 records consists of a Header record and any number (including 0) Line Break records (see JSDD Program Structure Section 4.5.1). ''

```
 item bytes integer $ '' no. of bytes proccessed''

 item cur'pos integer $ '' current position in line ''

 item done b $ '' termination flag ''

 item f1'rec integer $ '' the current FILE 1 record''

 item last'byte integer  $ '' byte pos of last break ''

 item last'rec integer $ '' rec of last break ''

 item stmt'type integer $ '' Part 2 statement  type  of  the
rec''

 item temp'byte integer $ ''used to store the break byte''

 item temp'rec integer $ '' used to store the break record''

proc create'h'ptr'rec (father=son) $
```

''The procedure CREATE'H'PTR'REC sets up pointers which have the effect of inserting an H'SON into the FILE 3 tree.''

```
 item father integer $ '' the father of the new record ''

 item son integer $ '' the record number of the new record''

proc create'v'ptr'rec (father=son) $
```

'' The procedure CREATE'V'PTR'REC is the same as CREATE'H'PTR'REC except that it creates a V'SON. ''

```
 item father integer $ ''record  number  of  the  new  rec's
father''

 item son integer $ '' record number of new FILE 3 rec ''

proc dashes (col,lngth,line,outpt) $
```

'' The procedure DASHES outputs a string of dashes of length LNGTH, starting on column COL of PUTOUT record LINE or TEMPC (depending on the value of OUTPT). ''

```
 item col integer $ '' the column on which dashes start ''

 item line integer $ ''PUTOUT record on which dashes
appear''

 item lngth integer $ '' length of dash string to be
output''

 item dash h 132 p
132h(-------------------------------------------
------------------------------------------------------------
----------------) $

 item outpt b $ '' output to TEMP or OUT'LINE ''

proc display'stump'ref (horiz) $

 ''The procedure DISPLAY'STUMP'REF creates and outputs a
stump reference display box either horizontally or
vertically from the current diagram code block. The box
contains the reference number of the stump which is
obtained from the PAGE'REF item of the stump's GROUP entry.
''

 item bottom integer $ '' the bottom line of the display
box''

 item display integer $ ''line on which reference appears''

 item horiz b $ ''type of display. horiz or vert.''

 item index integer $ '' the index of the stump's GROUP
entry''

 item left'start integer $ ''the starting column of the
box''

 item lngth integer $ '' used to store string lengths''

 item line integer $ '' displacement of lines on current
page''

 item page integer $ '' dummy variable for call to REMQUO ''

 item top integer $ ''the top line of box''
```

```
proc dots (col,lngth,line) $

 '' The procedure DOTS  outputs  a  string  of   LNGTH  dots
starting at column COL of the LINE th record of   PUTOUT. ''

 item col integer $ '' the starting column ''

 item line integer $ '' the number of the putout record''

 item lngth integer $ '' number of dots to be output ''

 item            dot            h            132            p
132h(.........................................
 ...............................................................
 ................) $


proc draw'line (from'x,from'y,to'x,to'y) $

 '' The procedure DRAW'LINE  draws a  line  from  the  point
having  x  and y coordinates (in terms of column and record)
FROM'X, FROM'Y to the point TO'X, TO'Y. ''

 item from'x integer $ ''starting x coord''

 item from'y integer $ '' starting y coord ''

 item line integer $ '' used to store page displacement ''

 item page integer $ ''dummy used in call to REMQUO''

 item to'x integer $ '' ending  x coord ''

 item to'y integer $ '' ending y coord ''

 item two'lines b  $''on when  2  lines  needed  to  connect
points''


proc extract'name () $

 '' The function EXTRACT'NAME  extracts and returns the name
of a  module  head. ''

 item done b $ '' termination flag ''

 item  extract'name character $ '' returns the module name''

 item index integer $ ''index into text string ''

 item index1 integer $ '' index into text string''
 item index'f2 integer $ '' FILE 2 record number''

 item lngth integer $ '' used to store lengths of  strings''

 item source'line character $ '' stores FILE 2 record text''


proc extract'text (f4'ptr) $

 '' The  function  EXTRACT'TEXT  extracts  FILE  2  text in
accordance with the line break information contained in  the
set  of   FILE  4 records which correspond to the statement
unit. ''

 item byte'ptr  integer  $  ''last  byte  of  text  to   be
returned''

 item  extract'text  character  $  ''contains  text  to  be
returned''

 item f4'ptr integer $ '' current FILE 4 record ''

 item field1 integer $ '' stores  F2'PTR of current  FILE  4
rec''

 item field2 integer $ ''stores LINES'OUT or F2'REC of rec''


 item  field3 integer $''stores MAX'LINE'LNGTH or F2'BYTE of
rec''
 '' LINE'PTR is the FILE 2 record which  contains  the  last
byte to be returned''

 item line'ptr integer $

 item lngth integer $ ''stores string lengths ''

 item next'f2 integer $ ''number of next FILE 2 rec''

 item next'f4 integer $ ''number of next FILE 4 rec''

 item text'line character $ ''temporary string storage''


proc generate'contents'entry $

 '' The procedure GENERATE'CONTENTS'ENTRY outputs a table of
contents entry for the current module. It finds the starting
page of the module in PAGE'REF.''
```

AD-A052 731 CHARLES STARK DRAPER LAB INC CAMBRIDGE MA F/G 9/2
JOVIAL STRUCTURED DESIGN DIAGRAMMER (JSDD). VOLUME III. PROGRAM--ETC(U)
FEB 78 G GODDARD, M WHITWORTH, E STROVINK F30602-76-C-0408
UNCLASSIFIED R-1120-VOL-3-PT-1 RADC-TR-78-9-VOL-3-PT-1 NL

2 OF 2
AD
A052 731

END
DATE
FILMED
5-78
DDC

```
 item  line  integer  $ '' LINE'NO's displacement on current
page''

 item lngth integer $ '' stores string lengths''

 item page integer $ '' dummy used in calling REMQUO''


proc generate'contents'header $

 ''The procedure GENERATE'CONTENTS'HEADER outputs the  page
heading  for the table of contents. ''

 item  contents'header  character  $ ''the table of contents
title''

 item lngth integer $ '' stores string lengths''


proc get'f1'rec   $

 ''The procedure GET'F1'REC tries to  read the next  FILE  1
record.  If an end of file is encountered, it sets EOFILE to
TRUE. ''

 item f1'ptr integer $ '' the number of the FILE 1 record ''



proc iformat (num) $

 ''  The function IFORMAT accepts an integer, and returns  a
character  string  (in  converted  form)  representing  the
integer.''

 item num integer $ '' the integer to be converted''

 item iformat character $ '' the string to be returned''

 item h6 h 6 $ '' a temporary for ENCODE''



proc init'block'constants $

 ''The  procedure  INIT'BLOCK'CONSTANTS  is called by Part 2
to  initialize  the  variables  which  will   be  constant
throughout  the creation of the code block. ''

 item  index  integer  $  ''index into f3'buf of the current
record''
```

proc initialize $

'' The procedure INITIALIZE sets up the initial values for DDG execution. ''

proc initiate'record (init'rec) $

'' The procedure INITIATE'REC sets starting values in a newly created FILE 3 record. ''

item init'rec integer $ '' rec no. of block being initiated. ''

proc insert (group'type,f3'index) $

'' The procedure INSERT links an entry into the GROUP table for a tree which is being inititated.''

item f3'index integer $ ''index of FILE 3 record being inserted''

item group'type integer $ ''type is module head or stump head''

proc legal'stump (stmt'type) $

'' The function LEGAL'STUMP accepts a code block type as input and outputs a 1 if the code block is a legal stump root. Otherwise, it returns a 0. ''

item legal'stump b $ '' holds the return value''

item stmt'type integer $ ''the input code block type''

proc max (int1,int2) $

'' The function MAX returns the larger of the two values passed to it. ''

item int1 integer $ ''input parameter''

item int2 integer $ ''input parameter''

item max integer $ '' return value''

```
proc min (int1,int2) $

 '' The  function  MIN returns the lesser of the two values
passed to it. ''

 item int1 integer $ ''input parameter''

 item int2 integer $ ''input parameter''

 item min integer $ ''the return value''


proc output'box'bottom $

 ''The procedure OUTPUT'BOX'BOTTOM outputs  the  bottom  of
the  code  block.  It  also  performs  some double buffering
optimization. ''

 item quo1 integer $ ''the page  on  which  the  box  bottom
appears''

 item  quo2  integer  $''the  page  on which V'SON's box top
appears''

 item rem1 integer $  ''dummy used in calling REMQUO''

 item rem2 integer $ '' dummy used in calling REMQUO''

 item x1 integer $ ''column of connecting line''

 item y1 integer $ ''start line of connector''

 item y2 integer $ '' end line of connector''


proc output'box'top $

 ''The procedure OUTPUT'BOX'TOP outputs the top line of  the
code block.''


proc output'header (page'top) $

 ''  The  procedure  OUTPUT'HEADER  outputs the page heading
starting on the line passed to it. ''

 item lngth integer $ '' stores string lengths''

 item page'top integer $ '' the line on  which  the  heading
starts''

 item temp'line character $ '' a tmp for character strings''
```

```
proc output'title'page $

'' The procedure OUTPUT'TITLE'PAGE outputs the title page
of the diagram. ''

 item line integer $ ''displacement on the current page''

 item lngth integer $ '' stores string lengths ''

 item page integer $ ''dummy used in calling REMQUO''

 item start'title integer $ ''page line on which title
starts''

 item temp'line character $ '' a character temp''

 item title'index integer $ '' the index into the title
array''

 item title'pages integer $ ''number of pages spanned by
titles''


proc part2 $

 '' The procedure PART2 is the driver routine for Part 2 of
the DDG. ''

 item lngth integer $ '' stores string lengths''

 item tmpi1 integer $ '' an integer temp''

 item tmpi2 integer $ '' an integer temp''


proc part2'init $

 '' The procedure PART2'INIT performs set-up tasks for Part
2 execution. ''

 item lngth integer $ '' stores string lengths''


proc ph2err (error'mess) $

 '' The procedure PH2ERR issues error messages and stops
execution of the DDG. ''

 item error'mess character $ '' the error message''
```

```
proc place (new'box) $

 '' The procedure PLACE  places the current  code  block  on
the design diagram. It also performs stump detection. ''

 item  bottom  integer  $  ''page  on  which  the box bottom
appears''

 item entrance integer $''vertical or horizontal entry  into
block''

 item  father'bottom  integer $ ''bottom line of father code
block''

 item father'left integer $ ''father's starting column''

 item father'right integer $ '' father's ending column''

 item father integer $ ''father's FILE3 record number''

 item father'type integer $ ''father code block's type''

 item horiz integer p  0  $  ''  constant  indicating  horiz
entry''

 item  mid  integer  $  ''  page  on  which block's midpoint
appears''

 item new'box integer $ ''new f3 rec describing  latest  box
''

 item  page'spans  integer  $  ''number  of pages spanned by
block''

 item rem1 integer $ '' page displacement''

 item rem2 integer $ ''page displacement''

 item rem3 integer $ '' page displacement''

 item stmt'type integer $ ''type of current FILE3 rec''

 item stump'ref'bottom  integer  $  ''bottom  of  stump  ref
display''

 item top integer $ ''page on which top of block appears''

 item  vert  integer  p  1  $ ''constant indicating vertical
entry''
```

```
proc pop'layout'info $

 '' The procedure POP'LAYOUT'INFO pops the  top  element  of
LAYOUT'STACK into LAST'LINE. ''


proc pop'proc'stack (=pop'rec) $

 '' The procedure POP'REC pops the top element of PROC'STACK
onto POP'REC. ''

 item pop'rec integer $ ''output parameter''


proc push'layout'info $
 ''  The  procedure  PUSH'LAYOUT'INFO  pushes  the  value of
LAST'LINE onto the top of LAYOUT'STACK. ''


proc push'proc'stack (push'rec) $

 '' The procedure PUSH'PROC'STACK pushes the number  of  the
current FILE 3 record onto the PROC'STACK. ''

 item push'rec integer $ '' the number of the record''


proc resolve'stump (stump'rec) $

 '' The procedure  RESOLVE'STUMP  finds the root  of a stump
and  invokes PLACE for each record currently hanging off the
stump root. ''

 item display'room b $ ''enough room for stump  ref  display
?''

 item done b $ ''completion flag''

 item father integer $ ''record number of father''

 item  horiz  integer  p  0  $  '' constant indicating horiz
entry''

 item index integer $ ''used as  an  index  into  the  GROUP
table''

 item init'stump integer $''record number of the rec causing
stump''

 item  last'index  integer  $  ''stores  previous index into
GROUP''
```

```
 item old'index integer $ ''previous GROUP index''

 item stump'rec integer $ '' current FILE3 record''

 item sub'stump integer $ ''record number of sub stump''

 item type integer p 0 $ '' horiz-vert flag''

 item vert integer p 1 $ '' constant indicating vert entry''


proc stars (col,lngth,line) $

 '' The procedure STARS outputs  a  string  of  LNGTH  stars
starting on column COL of the LINE th PUTOUT record. ''

 item col integer $ '' the column in which stars start''

 item  line  integer  $  ''the  PUTOUT record in which stars
appear''

 item lngth integer $ ''number of stars output''

 item             star             h              132            p
132h(***********************************************
*******************************************************************
*****************) $


proc transfer'write3 $

 '' The procedure TRANSFER'WRITE3 transfers the contents  of
the   most   recent  version of FILE 3 and the current buffer
contents into the spare version of FILE 3.''

 ''TMP'BUF is a temp buffer for FILE 3''
 table tmp'buf r f3'blksiz s n $
  begin
  item ent1 integer $
  item ent2 integer $
  item ent3 integer $
  item ent4 integer $
  item ent5 integer $
  item ent6 integer $
  item ent7 integer $
  item ent8 integer $
  item ent integer $
  item ent10 integer $
  item ent11 integer $
  item ent12 integer $
  item ent13 integer $
  item ent14 integer $
  end

 item tmpi integer $ '' temp store for RECS'IN'BLK3''


proc transfer'write4 $

 ''Same as TRANSFER'WRITE3 , only it operates on FILE 4.  ''

'' TMP'BUF is the temp FILE 4 buffer''
 table tmp'buf r f4'blksiz s n $
  begin
  item ent1 integer $
  item ent2 integer $
  item ent3 integer $
  end

 item tmpi integer $ ''temp store for RECS'IN'BLK4''


proc transfer'write'out $

 ''   Same   as   TRANSFER'WRITE3,   except   TRANSFER'WRITE'OUT
operates on PUTOUT. ''


proc update'file3  $

 ''The procedure UPDATE'FILE3 updates FILE 3  (by  adding  a
new record, or continuing an old one) in accordance with the
information in the newly created set of FILE 4 recs. ''

 item  la'type  integer  $  '' stmt'token of the next FILE 1
record''

 item new'rec integer $ ''the record number of a  new  FILE3
rec''

 item  stmt'type  integer $ '' the type of the current FILE3
rec''
```

4.6.3 DDG Compools

This section contains the DDG compools OPT and DEBUG. OPT contains the user options. OPT's variables are described in the JSDD USERS MANUAL. DEBUG contains debugging switches which are described in Section 7.2.2.

```
start $
 '' This is the options compool for instructions  ''
 '' in setting options , see JSDD Users' Manual. ''
common options $
 begin
 item display'delim b p 0 $
 item double'space b p 0 $
 item margin i 36 s p 5 $
 item mess'sw i 36 s p 1 $
 item page'lngth i 36 s p 60 $
 item page'width i 36 s  p 132 $
 item st'max i 36 s p 30 $
 item heading b p 1 $
 array header 10 h 150 $
 begin
 57h(c   s   draper   laboratory  jovial  structured  design
diagrammer)
 18h(DESIGN DIAGRAM OF )
 end
 item pgm'name h 150 p 21h(the design diagrammer) $
 item low'lim i 36 s p 20 $
 item max'width i 36 s p 40 $
 item name'index i 36 s p 1 $
 item head'no i 36 s p 1 $
 item table'of'contents b p 1 $
 item title'sw b p 1 $
 array title 70 h 150 $ begin
 1h( )
 1h( )
 1h( )
 41h(       this listing consists of output from)
 52h(       the charles stark draper laboratory's jovial j3)
 34h(       structured design diagrammer.)
 1h( )
 1h( )
 1h( )
 1h( )
 42h(       principal designers and implementors )
 1h( )
 37h(            gary w. goddard, csdl staff)
 39h(            mark h. whitworth, csdl staff)
 52h(            eric f. strovink, graduate  student,  m.i.t.)
 25h(computer science division)
 57h(the  charles  stark draper laboratory, inc., cambridge,
ma.)
 lh( )
 end
 item title'no i 36 s p 17 $ end term $

start $

common debug $
   begin
   item   debug1   b   p 1 $ ''task progress''
   item   debug2   b    p  0   $  ''access1  and  get'fl'rec
messages ''
   item   debug3   b   p 0 $ '' access2 ''
   item   debug4   b   p 0 $ ''access3 and transfer'write3''
   item   debug5   b   p 0 $ ''access4 and transfer'write4''
   item      debug6      b      p   1    $    ''access'out   and
transfer'write'out ''
   item   debug7   b   p 0 $ ''close''
   item   debug8   b   p 0 $ ''connect''
   item   debug9   b   p 0 $ ''construct & output'box'(top &
bottom)''
   item   debug10   b   p 0 $ ''continue and update''
   item   debug11   b   p 0 $ ''create'file4'recs''
   item   debug12   b   p 0 $ ''create'(h & v)'ptr'rec''
   item   debug13   b   p 0 $ ''display'stump'ref''
   item   debug14   b   p 0 $ ''draw'line''
   item   debug15   b   p 0 $ ''extract'n ame''
   item   debug16   b   p 0 $ ''extract'text''
   item   debug17   b   p 0 $ ''output'header''
   item   debug18   b   p 0 $ ''group progress''
   item   debug19   b   p 0 $ '' record progress''
   item   debug20   b   p 0 $ ''place''
   item   debug21   b   p 1 $ '' (push & pop)'proc stack  ''
   item   debug22   b   p 0 $ '' resolve'stump''
   item   debug23   b   p 0 $ '' contents entry''
   item   debug24   b   p 1 $ ''aviod buf span ''
   item   debug25   b p 0 $ ''no head stmp no.''
   item   debug26 b p 0 $''additional stump messages''
   item   debug27 b p 0 $ ''write blocks if error''
end
term $
```

4.7 Invocation Diagrammer

4.7.1 Introduction

The Invocation Diagrammer is a documentation tool which computes all possible procedure and function invocation trees for a given JOVIAL program. With the aid of an Invocation Diagram, one can examine an unfamiliar program and quickly determine which procedures are called from where. The experienced reader will appreciate this capability, since the first task in understanding any program is to gain some understanding of control flow. A glance at the DDG Invocation Diagram will demonstrate that this task is often non-trivial.

It is recommended that the User's Manual sections dealing with the Invocation Diagrammer be read and understood before continuing, since familiarity with the format of the diagrams is important to an operational understanding of the diagrammer.

Section 4.7.2 describes the general structure and operation of the diagrammer, Section 4.7.3 lists global variables, and Section 4.7.4 summarizes the procedures and local variables.

4.7.2 Operation and Program Structure

Operation of the diagrammer is fairly straightforward. First, procedure FIRST'PASS is called. FIRST'PASS reads in FILE 0 records and creates a sorted list of all procedure and function names it finds in the file. This list is stored in PROC'ARRAY. The main program, if there is one, is always at PROC'ARRAY(0). Array elements in FLAG'ARRAY, declared parallel to PROC'ARRAY, are set = 1 if the corresponding PROC'ARRAY entry is an internal procedure. This fact is computable from the FILE 0 format (see Section 4.4). Needless to say, FLAG'ARRAY is updated when a PROC'ARRAY update takes place, so that corresponding entries still match.

When FIRST'PASS completes, SECOND'PASS is invoked. SECOND'PASS fills a bit array (BIT'ARRAY) whose rows and columns both correspond to names in PROC'ARRAY. An element of BIT'ARRAY(a,b) is set if procedure PROC'ARRAY(a) calls procedure PROC'ARRAY(b). The FLAG'ARRAY element corresponding to PROC'ARRAY(b) is multiplied by 5000.

After SECOND'PASS, procedure WARSHALL is called, which computes the transitive closure of BIT'ARRAY. Procedure CHECK'RECURSION prints the names of any procedures which "call themselves" (have a nonzero bit flag at BIT'ARRAY(a,a), where a is the index into PROC'ARRAY of the procedure name). CHECK'RECURSION sets the FLAG'ARRAY element corresponding to these procedures = 10000 (all elements in BIT'ARRAY set by a closure operation are set by WARSHALL to "2", not "1", so that these "calls" will never be interpreted as "real" calls by future procedures).

After CHECK'RECURSION, the diagram is generated. This is done with a recursive procedure called PRINT. PRINT pushes an initial index into PROC'ARRAY(0), a corresponding starting column of BIT'ARRAY(1), an initial copy of HORIZ (which contains vertical diagram lines), and calls WRITE'PROC'NAME. WRITE'PROC'NAME writes out the procedure name, preserving all vertical diagram lines to the left. If the danger exists that an attempt might be made to write over the DDG page'width value, then a numbered "stump" is created. The stump character representation is appended to PROC'ARRAY, and appropriate new elements of BIT'ARRAY are set to effect the continuation. FLAG'ARRAY is adjusted so that this "procedure" looks like a procedure that is internal to the program unit, but never called by any other procedure. This ensures that it will print out at the end of the diagram.

If this new procedure name which has been pushed by PRINT calls another procedure, its index and BIT'ARRAY column index (index of the next procedure it calls) are pushed by PRINT, and the cycle continues. When PRINT pops a procedure name (when all the procedure's calls have been exhausted), the number of vertical diagram lines to the left is reduced by one. A pop at 0 lines means that the diagram is done. At this point, all procedures that were not called are printed as separate Invocation Diagrams, in the same way as the large diagram. Note that stumps will get printed at this point.

External procedures, detected because their names did not appear as the first record of any FILE 0 block, are flagged in the diagram with a "+". Procedures that were part of a recursive loop are printed normally (pushed) just once. Thereafter, they are printed as though they did not call any other procedures, and are flagged on the diagram with a "*". If this were not done, the diagram would print forever.

Page headings differ slightly from the convention in the DDG, because the diagrammer heading remains the same throughout the program (except for page numbers). It was felt that continuations would be rare, and isolated procedures few; thus, the extra code required to print a Table of Contents and vary the heading could be left out.

4.7.3 Global Variables

array bit'array 200 h 70$ '' contains a record of all procedure calls from all procedures ''

array column 200 integer$ ''part of information saved when a procedure name is pushed by print - indicates current column of horiz.''

item cont'flag integer p 0$ ''to determine whether to print continuation title line''

file file0 10000 r 301 v(a) v(b) v(c) v(d) v(eof) 11$ ''file0 declaration''

table file0b r 60 s n$ begin item f0'entry h 30 end ''used to read in file0 record''

array flag'array 200 integer$ ''parallel array to proc'array. Contains information about corresponding proc name in proc'array''

item gtemp70 h 70$ ''temporary''

item horiz h 150 p 1h( )$ ''contains vertical lines from previous section of diagram''

array host'proc 200 integer$ ''part of print-pushed info. Contains pushed procedure name''

item index integer$ ''contains index of proc name found in proc'array''

item line'length integer p 118$ ''same as page'width''

item line'number integer p 0$ ''current line number on output''

item max'entry integer p 0$ ''maximum index into proc'array''

array next'called 200 integer$ ''part of print-pushed info. Contains next proc called''

item page character p 5h(page)$ ''static constant''

array proc'array 200 h 30$ ''contains alphabetically

sorted list of all proc names encountered ''

item ptr integer p 0$ ''points at active set of push array information during recursion'' item rec'size integer$ ''used to input size of file0 record''

array stop'flag 200 integer$ ''part of print-pushed info. If set, we're done with this proc''

item stump'number integer p 0$ ''to keep track of stumps''

item temphh1 h 1$ ''temporary''

item temphh2 character$ ''temporary''

item temphh3 character$ ''temporary''

item temph30 h 30$ ''temporary''

item tempi1 integer$ ''temporary''

4.7.4 Procedures and Local Variables

proc out'line(line'contents)$

puts out one line into the output buffer

```
item line'contents character$ ''line to be put out''
item tempi1 integer$ ''temporary integer variable''
```

proc put'out(new'line)$

outputs a new line, adding a heading if necessary

```
item temph1 character$ ''temporary''
item temph2 character$ ''temporary''
item temph3 character$ ''temporary''
item tempi1 integer$ ''temporary''
item tempi2 integer$ ''temporary''
item temph6 h 6$ ''used for converting numbers''
item remainder integer$ ''temporary''
item page'number integer$ ''contains page number after
remquo''
item new'line character$ ''contains new line to be
output''
```

proc stump'no(=char30)$

purpose of proc is to return a stump character string of the form "----- < >" where < > is a number from one to 999. This string is then defined as another procedure, external and not called by any internal procs. Thus it will appear at the end of the diagram as a stump.

item char30 h 30$ ''contains stump string "-----<digits>"''
item h6 h 6$ ''used to convert stump number to character form''

proc numj(aa)$ begin

used to output a number during debugging phase. No longer utilized, but retained for future disaster.

item aa i 36 s$ ''number to be put out''
item bb h 6$ ''will contain character representation''

proc initialization$

proc sets up initial quantities, sets up title, clears bit array and flag arrays.

proc first'pass$ begin

this procedure reads in file0 and sorts proc and function names

item temp30 h 30$ ''temporary''
item tempil integer$ ''temporary''
item done b$ ''while loop flag''

proc find(temp30)$

returns index into proc'array of passed name

item find integer$ ''indicates whether temp30 has been found in proc'array''
item tempil integer$ ''temporary''
item temp30 h 30$ ''contains character string to be found in proc'array''
item search integer$ ''current guess''
item lower integer$ ''lower bound on search''
item upper integer$ ''upper bound on search''
item remain integer$ ''temporary for REMQUO''

proc insert(temp30)$

object of this procedure is to insert entries into proc'array

item temp30 h 30$ ''temporary''

proc equals(search,temp30)$

compares character strings on a bit level for sorting purposes. Returns 0 if arguments are equal (if indexed name in proc'array (indexed by search) is equal to argument string temp30), 1 if temp30 belongs after the index, and -1 if it belongs before.

```
item equals integer$ ''indicates whether temp30 is

above or below search''
item search integer$ ''index into proc'array''
item temp30 h 30$ ''character string to compare with
proc'array(search)''
item tempi1 integer$ ''temporary''
item counter integer$ ''used to implement bit compare''
item tempi2 integer$ ''temporary''
item temp301 h 30$ ''contains proc'array(search)''
```

proc second'pass$

sets up initial bit array for creating diagram

```
item done b$ ''while loop flag''
item tempi1 integer$ ''temporary''
item cur'proc integer$ ''index into proc'array of proc
under consideration''
item called'proc integer$ ''index into proc'array of
proc that was called by cur'proc''
item temp30 h 30$ ''temporary''
```

proc set'bit(aa,bb,cc)$

utility procedure to set a bit in the bit array database

```
item aa integer$ ''vertical index into bit'array''
item bb integer$ ''horizontal index iinto bit'array''
item cc integer$ ''value to set bit to''
item temp70 h 70$  ''temporary''
```

proc get'bit(aa,bb)$

utility procedure to read a bit in the bit array database

```
item aa integer$ ''vertical index into bit'array''
item bb integer$ ''horizontal index into bit'array''
item temp70 h 70$ ''temporary''
item get'bit integer$ ''contains value of specified
bit''
```

proc warshall$

    does a transitive closure on the bit array database

    item flag b$ ''while loop flag''

proc check'recursion$

    checks for recursive loops by examining main diagonal of bit matrix. If any element is set, we have a recursive loop.

    item flag b$ ''while loop flag''
    item message h 150$ ''temporary''

proc print$ begin

    this is a recursive proc which prints out the invocation diagram from the databases assembled by first'pass and second'pass

proc write'proc'name$

    this procedure does formatting of horiz and writes out a rightmost procedure name.

    item aa h 150$ ''temporary''
    item tempil integer$ ''temporary''
    item host'p h 30$ ''contains name of current proc (or, current "scope")''

proc write'horiz$

    this procedure dumps horiz based on the current column value

    item tempcl h 150$ ''temporary''
    item rl integer$ ''temporary''

proc push$

    this procedure pushes down the recursion stack of print by examining procedures called and determining if there are more to be dumped out beyond the current level

    item tempil integer$ ''temporary''
    item done b$ ''while loop flag''
    item new'proc integer$ ''new procedure name, next called procedure''

4.8 Compiling the JSDD

The best method of explaining the compilation of the JOVIAL Structured Design Diagrammer is to demonstrate the control cards which accomplish the task. The compool segments required for a complete compilation are:

spool -	contains string package declarations
data -	contains phase 1 global declarations
ntables -	contains phase1 parsing tables
utilities -	contains terminal I/O declarations
opt -	contains phase 2 and invocation diagrammer options
debug -	contains phase 2 debug switches

All programs to be compiled are assumed by the control card decks to be converted to GCOS file format. This is because MULTICS forces tabs stops to occur at 10 character intervals when it converts a MULTICS file for use by the GCOS Encapsulator. In order to circumvent this, files are explicitly converted before compilation such that tab stops occur at carriage positions 4,7,10,13, etc.

A "canned" deck is part of all the compilation decks - it is "invoked" by the select card:

```
$         select    >misc_libraries>jocit>compile
```

Expanded, this is really:

```
$         prmfl     **,r,r,>ml>jocit>jocit.032977
$         limits    10,49k,,20000
```

This arrangement of control cards always works correctly, except when there are no compools except "utilities". For some unknown reason, the deck fails in this instance. The solution is to concatenate the "etc" card contents to the "jovial" card, and re-submit.

4.8.1 JSDD Compilation Control Cards

Phase 1:

```
$         snumb     efs
$         ident     Strovink.5581c1412
$         jovial    name/ph18/,xref,map,nopt,lstou,
$         etc       poolin/11,12,13,14/,ncomdk
$         prmfl     11,r,s,utilities.cmp_out
$         prmfl     12,r,s,spool.cmp_out
$         prmfl     13,r,s,data.cmp_out
$         prmfl     14,r,s,ntables.cmp_out
$         print     p*
$         prmfl     c*,w,s,jovwrk>ph18.obj
$         select    >misc_libraries>jocit>compile
$         select    jovwrk>ph18.gcos -gcos
$         endjob

$         snumb     efs
$         ident     Strovink.5581c1412
$         jovial    name/synth/,xref,map,nopt,
$         etc       poolin/11,12/,ncomdk
$         prmfl     11,r,s,utilities.cmp_out
$         prmfl     12,r,s,data.cmp_out
$         print     p*
$         prmfl     c*,w,s,jovwrk>synth.obj
$         select    >misc_libraries>jocit>compile
$         select    jovwrk>synth.gcos -gcos
$         endjob
```

Phase 2:

```
$         snumb     efs
$         ident     Strovink.5581c1412
$         jovial    name/ph24/,xref,map,nopt,
$         etc       poolin/11,12,13,14/,ncomdk
$         prmfl     11,r,s,utilities.cmp_out
$         prmfl     12,r,s,spool.cmp_out
$         prmfl     13,r,s,opt.cmp_out
$         prmfl     14,r,s,debug.cmp_out
$         print     p*
$         prmfl     c*,w,s,jovwrk>ph24.obj
$         select    >misc_libraries>jocit>compile
$         select    jovwrk>ph24.gcos -gcos
$         endjob
```

Invocation Diagrammer:

```
$         snumb     efs
$         ident     Strovink.5581c1412
$         jovial    name/invoc/,xref,map,nopt,
$         etc       poolin/11,12,13/,ncomdk
$         prmfl     11,r,s,utilities.cmp_out
$         prmfl     12,r,s,spool.cmp_out
$         prmfl     13,r,s,opt.cmp_out
$         print     p*
$         prmfl     c*,w,s,jovwrk>invoc.obj
$         select    >misc_libraries>jocit>compile
$         select    jovwrk>invoc.gcos -gcos
$         endjob
```

5. Error Conditions

This section describes error conditions and associated messages which can occur during execution of the JOVIAL Structured Design Diagrammer. Section 5.1 discusses Phase 1 errors, Section 5.2 covers Phase 2 errors, and Section 5.3 deals with string package errors, which are common to both Phase 1 and Phase 2.

5.1 Error Conditions in Phase 1

Phase 1 errors are best represented in a tabular format - error message(s), cause(s), ramifications, and corrective actions.

Error:

```
MODIFIED PARSE
SKIPPED OVER TOKEN "  "
RESUMING
<  >  ::=  <  >  <  >
PARTIAL PARSE TO THIS POINT IS:
ILLEGAL SYMBOL PAIR:
```

Cause:

Phase 1 could not parse the input program.

Ramifications:

If this program was correctly parsed by the JOVIAL compiler without warnings, and does not contain JOVIAL J3X I/O constructs, then there is an error in the DDDG. The flowchart may or may not be affected.

Corrective actions:

Make sure that the program compiles without warnings. If this fails, contact implementors.

Error:

```
EOF AT INVALID POINT
ABORT ON BAD EOF
```

Cause:

Phase 1 has encountered an END OF FILE on the input source deck before finding a compilable program unit.

Ramifications:

Can be the result of a parsing error, if certain crucial tokens get skipped earlier on. Other messages should precede these if this is the case. The flowchart will be incorrect.

Corrective actions:

The input file structure should be carefully examined for abnormalities.

Error:

MACRO TABLE OVERFLOW
SYMTAB OVERFLOW
MACRO DEFINITION TABLE OVERFLOW
FSTACK OVERFLOW

Cause:

An internal table has overflowed.

Ramifications:

All but SYMTAB OVERFLOW are fatal errors. SYMTAB OVERFLOW will affect only the invocation diagram.

Corrective actions:

Recompile Phase 1 with larger table sizes, or:
- MACRO TABLE OVERFLOW - reduce number of program macros
- MACRO DEFINITION TABLE OVERFLOW - see above
- FSTACK OVERFLOW - reduce nest level of function calls
- SYMTAB OVERFLOW - reduce number of proc/function calls

Error:

GETCRD MACRO BUFFER OVERFLOW

Cause:

A recursive macro definition has occurred, or macro nest level has exceeded approximately 25.

Ramifications:

Error is fatal.

Corrective actions:

Remove recursive definition, or reduce nest level.

Error:

ILLEGAL CHARACTER IGNORED

Cause:

An illegal character was detected in the JOVIAL source input

Ramifications:

Character is ignored; flowchart is unaffected.

Corrective actions:

Remove character.

5.2. DDG Error Conditions and Debugging Messages

This section contains a description of errors which are reported by the DDG and DDG debugging switches which can be used to locate DDG malfunctions. Section 5.2.1 lists DDG error conditions and Section 5.2.3 describes the DEBUG switches.

5.2.1 DDG Error Conditions

The DDG reports occurrences of twelve types of errors. All DDG errors are fatal. The following is a list of error messages, their meanings and in some cases, possible corrective actions.

PROC STACK ERROR
Part 2 processing has completed, but all entries pushed onto the PROC'STACK have not been popped off. The message is issued by the main routine at a time when the PROC'STACK should be empty. Occurrence of this error indicates an error in either the DDG execution or in FILE 1.

GROUP STORE OVERFLOW
The storage capacity of the GROUP table has been exceeded. To overcome this problem, the DDG should be recompiled with a larger value for GROUP'MAX and more entries specified in GROUP's declaration.

BACK PTR ERROR
A FILE 3 record which requires a non-zero BACK'H or BACK'V field has none. An attempt to back through the record (in PLACE) has aborted. Occurrence of this error indicates a DDG failure.

PROC OR STUMP HANDLING ERROR
An attempt has been made to pop an element off an empty LAYOUT'STACK. Occurrence of this error indicates a DDG failure.

CONTROL HEAD ERROR
The name of a control head was expected (by EXTRACT'NAME) but not found. Occurrence of this error indicates an error in either the DDG execution or in FILE 2.

PROC HANDLING ERROR
An attempt has been made to pop an element off of an empty PROC'STACK. Occurrence of this error indicates a DDG failure.

LAYOUT STACK OVERFLOW
The capacity of the LAYOUT'STACK has been exceeded. The sizes of LAYOUT'STACK and LAYOUT'STACK'MAX should be increased.

PROC STACK OVERFLOW
The capacity of PROC'STACK has been exceeded. Increase the sizes of PROC'STACK and PROC'STACK'MAX.

STUMP HANDLING ERROR
A stump was detected during RESOLVE'STUMP's attempt to find a legal stump root. This error indicates a DDG failure.

STUMP HANDLING ERROR 2
A stump was found during RESOLVE'STUMP's attempt to find a stump root whose father's placement could accomodate a stump reference display. Occurrence of this error indicates a DDG failure.

STUMP HANDLING ERROR 3 REC n
Record n (FILE 3) was detected to be an illegal stump during RESOLVE'STUMP's invocation of PLACE. This error indicates DDG failure.

SUB STUMP SEARCH FAILURE
A sub stump was detected but not found in the GROUP table. Occurrence of this error indicates DDG failure. (A sub stump is a previously processed stump which is encountered again because of the backup necessary in finding the root of the current stump).

5.2.2 DDG DEBUG Switches

The DDG is run with a compool DEBUG (see Section 4.6.3) which contains 26 DEBUG switches which control the messages that can be used to monitor the progress of the DDG's execution.

Under normal conditions all DEBUG switches should be turned off (i.e. preset to zero). However, should a DDG failure occur, the DEBUG switches can be used to localize the DDG failure.

The destination of the DEBUG messages is controlled by the user option MESS'SW (see JSDD USER'S MANUAL). Messages can be sent either to the user's terminal or to the file whose device number is 12.

The following is a list of the DEBUG switches accompanied by a brief description of the messages that they control.

DEBUG1 controls the emission of messages that are general progress reports of DDG execution. The messages are:

INIT COMPLETE
: Execution of the procedure INITIALIZE has been completed.

END PART 1
: Part 1 execution has completed.

COMPUTE NUMBERS
: The procedure COMPUTE'PAGE'NUMBERS has been invoked.

CONTENTS HEADER
: The page heading for a table of contents page has been output.

PART 2 DONE
: Part 2 execution has completed.

LAST LINE n
: The diagram will consist of n lines of output.

START BYTING
: The output truncating procedure BYTE'EM has been invoked.

DEBUG2 controls the emission of messages concerning the FILE 1 interface routines (ACCESS1 and GET'F1'REC). Output is voluminous. Messages are:

ACCESS1 m
: The mth FILE 1 record is being accessed.

ACCESS1 GET n
: The nth block of FILE 1 is being read.

FILE 1 EOF
The end of FILE 1 has been encountered.

DEBUG3 controls the emission of messages concerning the FILE 2 interface routine (ACCESS2). Output is voluminous. The messages are:

ACCESS2 m
The mth FILE 2 record is being accessed.

ACCESS2 GET n
The nth FILE 2 block is being read.

DEBUG4 controls the emission of messages concerned with the FILE 3 interface routines (ACCESS3 and TRANSFER'WRITE3). Output is voluminous. The messages are:

READ3 m
The mth record of FILE 3 is being accessed.

TRANS-OUT n
The nth FILE 3 block is being written out to disk.

DEBUG5 controls the emission of messages concerned with the FILE 4 interface routines (ACCESS4 and TRANSFER'WRITE4). Output is voluminous. The messages are:

READ4 m
The mth FILE 4 record is being accessed.

TRANSFER4-out n
The nth FILE 4 block is being written to disk.

DEBUG6 controls the emission of messages concerned with PUTOUT's interface routines (ACCESS'OUT and TRANSFER'WRITE'OUT). The messages are:

ACCESSOUT GET m
The mth block of PUTOUT is being read.

TRANS OUT n
The nth PUTOUT block is being written to disk.

MAX FOUTPUT p
PUTOUT now has a total of p blocks.

DEBUG7 is not used.

DEBUG8 controls messages emitted in the procedure CONNECT'BOXES. Output is voluminous. The message is:

CONNECT m TO n
Boxes m and n are being connected.

DEBUG9 controls the emission of messages concerned with outputting code blocks. Output is voluminous. The messages are:

CONSTRUCT
: The procedure CONSTRUCT'LINE has been invoked. The line (of the JSDD) being constructed is also output.

TOP OUT
: The procedure OUTPUT'BOX'TOP has been invoked.

BOTTOM OUT n
: The bottom line of the nth code block is being output.

DEBUG10 controls the emission of messages concerning FILE 3 record creation and continuation. Output is voluminous. The messages are:

UP END SCOPE
: An END'SCOPE was encountered in FILE 1.

BACK THRU m
: The FILE 3 tree is being backed through in an attempt to find the last control phrase. Record m is the current FILE 3 record.

UP CONTROL1
: A CONTROL1 record is being created.

UP SCOPE START
: A FILE 3 record beginning the scope of a control phrase is being created.

UP NEW TYPE
: A new FILE 3 record of a different type than the last is being created.

CONT n
: The nth FILE 3 record is being continued.

DEBUG11 controls the emission of messages concerning FILE 4 record creation. Output is voluminous. The messages are:

CREATE4 m
: The mth FILE 4 record is being created.

END CREATE n
: The nth FILE 4 record was the last created in the current set.

DEBUG12 controls the emission of messages concerned with the creation of sons of FILE 3 records. The messages are:

CREATE H m
: The mth FILE 3 record is being created by CREATE'H'PTR'REC. It is accessed by an H'PTR.

CREATE V n
: The nth FILE 3 record is being created by CREATE'V'PTR'REC. It is accessed by a V'PTR.

DEBUG13 controls the emission of a message concerned with the displaying of stump references. The message is:

  DISP STUMP m

    The stump whose father is record m is having its stump reference display output.

DEBUG14 controls the emission of messages output by the procedure DRAW'LINE. Output is voluminous. The messages are:

  START'DRAW

    DRAW'LINE has been invoked. The coordinates of the endpoints of the line to be drawn are also output.

  END DRAW

    The execution of DRAW'LINE has been successfully completed.

DEBUG15 controls the emission of messages output in the procedure EXTRACT'NAME. The message consists of:

  EXTRACT NAME

    EXTRACT'NAME has been invoked. Following this message, the extracted name is output.

DEBUG16 controls the emission of messages relating to the text extraction routine (EXTRACT'TEXT). Output is voluminous. The message is:

  EXTRACT TEXT m

    EXTRACT'TEXT has been invoked and will operate on FILE 4's mth record. The extracted text is also output.

DEBUG17 controls the emission of a message concerned with outputting page headings. Output is voluminous. The message is:

  PAGE'TOP m

    The heading for the page that starts on line m is being output.

DEBUG18 controls the emission of a message reporting on the progress of threading through the GROUP table. The message is:

  START GROUP m

    Part 2 processing has begun on the FILE 3 tree pointed to by the mth entry of GROUP.

DEBUG19 controls the emission of a message reporting on the progress of Part 2's code block generation. Output is voluminous. The message is:

  CUR'REC m

    The mth code block is being output.

DEBUG20 controls the emission of messages concerning Part I code block placement. Output is voluminous. Messages are:

PLACE m

The procedure PLACE is operating on the mth code block.

SPANS

The mth code block spans a page heading.

NO SPAN

No page heading is spanned.

DEBUG21 controls the emission of messages concerning operations performed on the PROC'STACK. The messages are:

PUSH m

The record number m is being pushed onto the PROC'STACK.

POP OFF m

The record number m is being popped off of the PROC'STACK.

DEBUG22 controls the emission of messages output by the procedure RESOLVE'STUMP. The messages are:

RESOLVING STUMP n

FILE 3 record n has been found to be the root of the stump.

DEBUG23 controls the output of entries in the table of contents. The message is:

TABLE ENTRY

The procedure GENERATE'CONTENTS'ENTRY has been invoked. The name of the module whose table entry is being output is also printed.

DEBUG24 controls the emission of a message output during the buffer optimization operation performed by OUTPUT'BOX'BOTTOM. The message is:

AVOID BUF SPAN m AND n

Part of the connecting line between code blocks m and n is being drawn now. See Appendix C.

DEBUG25 controls the emission of messages concerned with assigning sequential reference numbers to stumps. Sequential reference numbers are assigned only when the HEADING option is not in effect. The message is:

STUMP NO m

The mth stump has had its reference number assigned.

DEBUG26 controls the emission of messages concerned with stump resolution. The messages are:

STUMP CAUSE REC m
The mth FILE record caused the placement failure which resulted in this call to RESOLVE'STUMP.
REORDER STUMP p
The FILE 3 record p has been found to be a sub stump. Its position in GROUP is being changed so that it follows the stump which is rooted by record n.
RESOLVE:TREE END q
FILE 3 record q is the leaf (i.e., it points to no FILE 3 record) of the current stump tree.

DEBUG27 causes FILES 3 and 4 to be written to disk if a DDG error is detected. A message is output indicating which versions of the files are the most complete.

5.3 Error Conditions in String Package

Error:

*** CONCAT ERROR ***
*** SUBSTR ERROR ***
*** SPACES ERROR ***

Cause:

The String Package has detected an error in the use of extended string functions. It issues a warning, and performs a cleanup action as described in Section 4.1.4.

Ramifications:

The flowchart will not be affected, as the cleanup of these errors is well-structured and logical.

Corrective Actions:

There is an internal error in the program calling the string package. The most likely sources of such errors are the DDDG SCAN routine and the DDG EXTRACT'TEXT routine. Contact implementors if these errors occur frequently.

6. Operator Instructions

No special instructions need be directed to the computer operator.

Section 7 Printout

This section consists of printout. It is printed separately as Part II of this volume because of its' bulk.

Preceding Page Blank

Section 8 Printout

This section consists of printout. It is printed separately as Part III of this volume because of its' bulk.

Section 9 Printout

This section consists of printout. It is printed separately as Part IV of this volume because of its' bulk.

APPENDIX A Representing Programs as Binary Trees

The representability of computer programs as binary trees is fundamental to the operation of the JSDD.

Any structured program written in JOVIAL can be represented as a finite binary tree. There are ,however, non-structured JOVIAL programs which cannot be so represented. Any JOVIAL program (structured or not) can be represented as a finite graph (not necessarily a tree) or as a binary tree (but not necessarily a finite tree).

Consider a non-structured compound statement of the following form:

```
    BEGIN
          CODE BLOCK A
          LABEL1. IF CONDITION $
                BEGIN
                CODE BLOCK B
                GOTO LABEL1 $
          END
          CODE BLOCK C
```

A finite graph representing the compound statement is:

```
***************
*CODE'BLOCK'A $*
****************
:
:
:
:<-------------------------------------+
:                                      :
V                                      :
*****************     *****************:
*IF CONDITION $  *----->*CODE'BLOCK'B $ *---+
*****************     *****************
:
:
:
:
V
*****************
*CODE'BLOCK'C $ *
*****************
```

A diagrammer employing this representation would produce unreadable diagrams for programs having any degree of complexity.

The JSDD's representation of the compound statement is:

```
****************
*CODE'BLOCK'A $ *
****************
 '

 '

**************************        ****************
*LABEL1 . IF CONDITION $  *------*CODE'BLOCK'B $ *
**************************        *GOTO LABEL1 $  *
 '                                ****************
 '
 '
 '
****************
*CODE'BLOCK'C $ *
****************
```

The JSDD diagram does not clearly illustrate the fact that evaluation of CONDITION is repeated. The JOVIAL structured extensions were introduced for this reason. They eliminate the need for using non-structured constructs. The code in the example compound statement should be rewritten in the form:

```
CODE BLOCK A
DO WHILE [CONDITION]
     CODE BLOCK B
[END DO]
CODE BLOCK C
```

The JSDD representation of the rewritten code is:

```
****************
*CODE'BLOCK'A $ *
****************
 '
 '
 '
 '
**************************        ****************
*DO WHILE CONDITION $     *------*CODE'BLOCK'B $ *
**************************        ****************
 '
 '
 '
 '
****************
*CODE'BLOCK'C $ *
****************
```

The repeated evaluation of CONDITION is clearly shown in the control box (i.e. the box containing CONDITION).

Consider a more complicated compound statement :

```
BEGIN
     CODE'A
     IF COND1 $
          BEGIN
          IF COND2 $
               BEGIN
               IF COND3 $
                    BEGIN
                    CODE'B
               END
               CODE'C
          END
     END
     CODE'D
END
```

The JSDD representation of this compound statement is :

```
***********
*CODE'A $ *
*''1''    *
***********
 '
 '
************
*IF COND1 $ *    ************
*            *---*IF COND2 $ *    ************
*''2''      *    *            *---*IF COND3 $ *     ***********
************     *''3''      *    *             *---*CODE'B $ *
 '               ************     *''4''      *     *''5''    *
 '                                ************      ***********
 '                                 '
 '                                 '
 '                                ***********
 '                                *CODE'C $ *
 '                                *''6''    *
 '                                ***********
 '
***********
*CODE'D $ *
*''7''    *
***********
```

The quoted numbers in each box indicate the order in which the tree nodes are always processed.

The traversal of JSDD binary trees is accomplished by simulated recursion. The JSDD trees are regarded as collections of subtrees related in the following manner:

```
******       ***********
*ROOT *----> *RIGHT SON*
******       ***********
   :
   V
**********
*LEFT SON*
**********
```

where LEFT'SON and RIGHT'SON are roots of the LEFT and RIGHT subtrees.

A recursive algorithm for the traversal might be written:

```
    RECURSIVE PROC TRAVERSE (ROOT) $
        BEGIN
        VISIT (ROOT) $
        TRAVERSE (RIGHT'SON) $
        TRAVERSE (LEFT'SON) $
    END
```

VISIT is the procedure which operates upon each node of the tree.

The simulation of the recursive procedure TRAVERSE is implemented with a stack called TRAVERSE'STACK. The simulation is presented on the following page.

```
PROC TRAVERSE (ROOT) $
    ITEM ROOT I 36 S $
    ITEM STACK'TOP I 36 S P 0 $
    ARRAY TRAVERSE'STACK 100 I 36 S $
BEGIN
    ''INITIALIZE STACK''
    TRAVERSE'STACK ($STACK'TOP$) = 0 $
    DO WHILE [ ROOT GQ 0 ]
        ''WHEN ROOT IS ZERO , ALL NODES OF''
        ''THE TREE HAVE BEEN VISITED''

        STACK'TOP = STACK'TOP + 1 $
        ''STACK ROOT  SO WE CAN RETRIEVE IT LATER ''
        TRAVERSE'STACK ($STACK'TOP$) = ROOT $
        VISIT (ROOT) $
        IFEITH RIGHT'SON NQ 0 $
            '' IT HAS A RIGHT SON.''
            ''TRAVERSE ITS SUBTREE''
            ROOT = RIGHT'SON $
            ORIF LEFT'SON NQ 0 $
                 '' HAVE A LEFT'SON.''
                 ''POP ITS FATHER AND TRAVERSE THE SUBTREE ''
                 BEGIN
                 ROOT = LEFT'SON $
                 STACK'TOP = STACK'TOP - 1 $
            END
            ORIF 1 $
                 ''THE CURRENT ROOT IS A LEAF.''
                 ''LOOK BACK FOR A LEFT SUBTREE TO TRAVERSE ''
                 BEGIN
                 DO WHILE [LEFT'SON EQ 0 AND ROOT NQ 0 ]
                      ROOT = LEFT'SON $
                      STACK'TOP = STACK'TOP - 1 $
                 [ END DO ]
            END
            END
       [END DO]
  END
```

APPENDIX B Statement Units, Statement Tokens and Mappings

This appendix describes the reductions made by Phase 1 of the JSDD which generate statement tokens and/or statement units. The list is ordered according to the values of STMT'TOKEN. To the right of the STMT'TOKEN, the reductions that generate the STMT'TOKEN are listed.

STMT'TOKEN	REDUCTIONS
1 END'SCOPE	<CLOSE DECLARATION> <PROCEDURE DECLARATION> <THEN CLAUSE> <ALTERNATIVE STATEMENT> <COMPLETE LOOP> <INCOMPLETE LOOP> <STRUCTURED EXTENSION> <CASE> <PROGRAM TAIL>
2 COMMENT'1	No reduction. This STMT'UNIT refers to a same-line (or Type-1) comment.
3 DELIM'COMMENT	Same as Comment'1, except that it modifies a BLOCK DELIMETER.
4 BLOCK'BEGIN	<BEGIN>
5 BLOCK'END	<END>
6 END'DO	<END DO>
7 END'CASE	<END CASE>
8 PGM'TAIL	<PROGRAM TAIL>
9 DEF'DIR	<DEFINE DIRECTIVE>
10 MODE'DIR	<MODE DIRECTIVE>
11 COM'HEAD	<COMMON HEAD>
12 SWITCH'DEC	<SWITCH DECLARATION>
13 PGM'DEC	<PROGRAM DECLARATION>
14 SPEC'PART	<SPECIAL PART>
15 TST'STMT	<TEST STATEMENT>
16 IO'STMT	<IO STATEMENT>
17 DIR'STMT	<DIRECT STATEMENT>
18 ASSGN'STMT	<ASSIGNEMENT STATEMENT>
19 EXCHNG'STMT	<EXCHANGE STATEMENT>
20 RETURN'STMT	<RETURN STATEMENT>
21 STOP'STMT	<STOP STATEMENT>
22 PROC'CALL	<PROCEDURE CALL>
23 MOD'DEC	<MONITOR DECLARATION>
24 FILE'DEC	<FILE DECLARATION>
25 ITEM'DEC	<SIMPLE ITEM DECLARATION>
26 ARRAY'DEC	<ARRAY DECLARATION>

STMT'TOKEN	REDUCTIONS
27 ORD'HEAD	<ORD HEAD>
28 ENT'DEC	<ENTRY ITEM DEC>
29 DEF'HEAD	<DEF HEAD>
30 DEF'DEC	<DEF ITEM DEC>
31 STRING'DEC	<STRING ITEM DEC>
32 LIKE'DEC	<LIKE TABLE DEC>
33 IND'OVER	<INDEPENDENT OVERLAY DEC>
34 SUB'OVER	<SUBORDINATE OVERLAY DEC>
35 BEGIN2	<BEGIN2>
36 END2	<END2>
37 GOTO'STMT	<GOTO STATEMENT>
38 COMMENT'2	No reduction. A type-2 (or C-type) comment.
39 PGM'HEAD	<PROGRAM HEAD>
40 CLOSE'HEAD	<CLOSE HEAD>
41 PROC'DESC	<PROCEDURE DESCRIPTOR>
42 IF'CLAUSE	<IF CLAUSE>
43 INC'FOR	<INCOMPLETE FOR>
44 FOR'CLAUSE	<FOR CLAUSE>
45 DO'HEAD	<DO HEAD>
46 ORIF'CLAUSE	<ORIF CLAUSE>
47 INSTANCE	<INSTANCE>
48 IFEITH'COND	<IFEITH CLAUSE>
49 IFEITH'HEAD	<IFEITH>
50 CASE'HEAD	<CASE HEAD>

Note that the reduction to <PROGRAM TAIL> causes two STMT'TOKEN values to be emitted by Phase 1 of the JSDD. These are PGM'TAIL (8) and END'SCOPE (1). They are emitted in that order to allow Phase 2 to first print out the <PROGRAM TAIL> and then end the scope of the <PROGRAM HEAD>.

In Phase 2 of the JSDD, the values of STMT'TOKEN are mapped onto a set of STMT'TYPE (or STMT'UNIT) values. The mapping is performed by the function BOX'MAP. The mapping is defined below.

I	BOX'MAP(I)	
1	END'SCOPE	
2	COMMENT'1	
3	DELIM'COMMENT	If DELIM'DISPLAY is off.
	COMMENT'1	If DELIM'DISPLAY is on.
3<I<8	BLOCK'DELIM	If DELIM'DISPLAY is off.
	SEQ	If DELIM'DISPLAY is on
8	PGM'TAIL	

8<I<38	SEQ
38	COMMENT'2
38<I<42	CONTROL'1
41<I<46	CONTROL'2
45<I<49	CONTROL'3
48<I<51	CONTROL'4

Of these values, only COMMENT'1, SEQ, PGM'TAIL, COMMENT'2, CONTROL'1, CONTROL'2, CONTROL'3, and CONTROL'4 are printing STMT'TYPEs. The DISPLAY'DELIM option is handled by BOX'MAP in that the value returned by BOX'MAP determines whether a statement unit will appear on the design diagram.

See Section 4.5.3.1 for a description of the STMT'TOKEN code block formats.

APPENDIX C Optimizing the Double Buffering System

Double buffering is one of the most time consuming operations which takes place in Phase 2 of the JSDD. Future versions of the JSDD will eliminate much of the expense of simulating direct access file facilities. The current JSDD contains two optimizations of the output file double buffering.

The first optimization is the elimination of invocations of the procedure DRAW'LINE (see Section4.5.3.2) where the line to be drawn spans a block boundary. This optimization is performed in the procedure OUTPUT'BOX'BOTTOM (see Section 4.5.3). After the bottom line of a code block is output to PUTOUT (the output file), its V'PTR is examined. If V'PTR is greater than zero, then the current code block will have another code block appearing beneath it (i.e., its son). If the top line of the son appears in another block of PUTOUT records, then a connecting line is drawn from the bottom of the current code block to the last record in the block currently in core. The STOP'LINE field of the current code block's FILE 3 record is then reset to the record number of the last record in core. This operation permits the connecting line between the current code block and its son to be drawn in two sections -- neither of which spans the block boundary.

The second optimization of the double buffering system involves an extra pass through the output file (in the procedure BYTE'EM). BYTE'EM loops through PUTOUT's records and calls the BYTE function which truncates the records at PAGE'WIDTH characters. It is more efficient to defer the truncation of PUTOUT's records until their generation is completed because it requires only one truncation per record. If truncation was performed during PUTOUT's generation, multiple truncations of each record would be necessary.

Future versions of the JSDD will include optimizations that will completely eliminate the need to double buffer PUTOUT. These optimizations will implement a queue which will store the progress of a diagram branch's output processing. When a branch reaches a PUTOUT block boundary its processing will be suspended and all other branches which use the PUTOUT block currently in core will be processed until the end of the buffer is reached. When all of the diagram branches have been processed in this manner, the buffer will be permanently written to disk, and the suspended branches will resume processing in the next block of PUTOUT.